CHALMERS

Efficient Data Structures in a Lazy Functional Language

Martin Holters

Department of Computing Science
Chalmers University of Technology
Göteborg, Sweden 2003

Abstract

Although a lot of theoretical work has been done on purely functional data structures, few of them have actually been implemented to general usefulness, let alone as part of a data structure library providing a uniform framework.

In 1998, Chris Okasaki started to change this by implementing Edison, a library of efficient data structures for Haskell. Unfortunately, he abandoned his work after creating a framework and writing some data structure implementations for parts of it.

This document first gives an overview of the current state of Edison and describes what efficiency in a lazy language means and how to measure it in a way that trades off complexity and precision to produce meaningful results.

These techniques are then applied to give an analysis of the sequence implementations present in Edison. Okasaki only briefly mentions the main characteristics of the data structures he has implemented, but to allow the user to choose the most efficient one for a given task, a more complete analysis seems needed.

To round off Edison's sequence part, four new implementations based on previously known theoretical work are presented and analysed: two deques based on the pair-of-lists approach, and two data structures that allow constant time appending, while preserving constant time for `tail` and, for one of them, even `init`.

To achieve a certain confidence in the correctness of the new implementations, we also present QuickCheck properties that not only check the operations behave as desired by the abstraction, but also allow data structure specific invariants to be tested, while being polymorphic.

Preface

This is a thesis for the degree of Master of Science. It documents a project aimed at improving Edison by analysing existing data structure implementations and adding new ones.

The reader is assumed to be familiar with functional programming and lazy evaluation in general and Haskell in particular. Furthermore, basic knowledge in amortised runtime analysis is recommended, although the techniques used in this thesis will be outlined briefly. To fully understand the QuickCheck related code fragments, the QuickCheck manual should be read. But mostly, the meaning of the used QuickCheck constructs is quite apparent.

This thesis can also be used as a companion to the Edison manual, where mostly the introductory overview and the time-complexity overview in the appendix are relevant.

My thanks go to my supervisor Koen Claessen, who not only proposed this project to me and provided excellent guidance, but had also gotten me excited about functional programming in the first place.

Contents

Chapter 1

Introduction

For most modern imperative programming languages, there exist ready-to-use libraries of efficient data structures for sets, bags, finite maps, priority queues etc. The Java Collection Classes and the Standard Template Library for C++ being prominent examples. Furthermore, most textbooks on data structures that claim to be language-neutral silently assume an imperative language. But when a functional programmer needs a certain data structure, he often has to browse through a multitude of papers and only with luck, he will also find an implementation that suits his needs. Motivated by this, Chris Okasaki published his book on functional data structures in 1998 [Oka98] and started the development of Edison, a functional data structure library written in Haskell.

Unfortunately, Okasaki abandoned his work on Edison after he had created a framework and written a few implementations. The aim of this thesis is to complement his work by giving a more detailed description of the implementations than what is found presently in the Edison manual and enrich the library with more implementations. In particular, after we briefly outline the methods used to perform runtime complexity analysis and describe the current status of the Edison library, a detailed time complexity analysis will be carried out in chapter 2 for the existing implementations. This is necessary as the papers describing the implemented data structures as well as the Edison manual are limited to only discuss the main operations, but the implementations in Edison provide many more. Due to the limited scope of a master thesis, we will however be limited to only a certain part of Edison, namely `Sequence`s. For the same, four new implementations are introduced in chapter 3: `SimpleDeque`, `BankersDeque`, `CatenableList` and `CatenableDeque`. Again, the papers on which these implementations are based only describe a minority of the operations available in Edison, the remaining functions were developed during the course of this project.

To ensure a certain quality of the implementations in Edison, the QuickCheck package developed by Claessen and Hughes [CH00] is used. It allows the specification of properties as Haskell functions and automatic testing of these properties for a large number of randomly generated test cases. In order to automatically generate random data of a certain type, that type has to be an instance of the `Arbitrary` class, which mainly defines an `arbitrary` function responsible of generating random data while observing required structural invariants. This is done inside the `Gen` monad defined as part of QuickCheck, which provides several useful combinators. At present, Edison contains property definitions for part of the defined type classes. All instances of these also provide the `Arbitrary` interface to allow automatic testing. However, the way the QuickCheck properties are currently specified has some drawbacks, so we develop our own in section 3.1.

Finally chapter 4 concludes by comparing Edison to other data structure libraries, both functional and object-oriented, discussing the differences and proposing work to be done in the future to make Edison a useful library for Haskell programmers.

1.1 Time Complexity in a Lazy Language

In this document, we will often worry about the runtime complexity of a function. But measuring time complexity in a lazy language is a little tricky. Not only is it difficult to determine when a function is actually evaluated, but furthermore, it may well be evaluated only partially. Wadler [Wad88] and Sands [San90] have developed methods to describe this precisely. However, their machinery seems to be too complex in a situation like this, where a programmer wants to figure out easily which implementation is best-suited for a certain purpose. Instead, one single term using $\mathcal{O}$-notation per function seems more appropriate, even at the cost of precision.

The easiest way to produce such a term is to analyse the function pretending the implementation language was strict. Unfortunately, many efficient functional data structures rely on laziness algorithmically, which prevents this approach. The important insight is that they only rely on laziness for constructs used *internally.* Thus, it is reasonable to simplify the time complexity analysis by assuming strictness for all function arguments or results that are of a different data type than the analysed structure. For the parameters, this is the only reasonable assumption; otherwise, forcing them could yield arbitrarily long runtimes. For the return value, this is equally apparent for atomic values: If they were not needed, the function would not be evaluated at all, making it unnecessary to worry about its runtime.

Only for composite return values (notably lists), this simplification actually discards useful information, namely whether the function is monolithic or at least partially incremental. Here we sacrifice precision for the sake of brevity.

Under these assumptions, amortised runtimes can be determined with the methods also used by Okasaki [Oka95a, Oka98], who usually describes his data structures in an environment where laziness can be applied selectively (i.e. Standard ML). The only situation that may require some additional thought is the usage of functions returning a different composite type to implement another function as part of the library. (A typical scenario for this is an implementation of a specific function that converts the data structure to a list, operates on the list and converts the result back.) Here, the incrementality neglected before may be relevant.

We will briefly outline the methods for amortised runtime analysis used throughout this document, while a more detailed description is left to [Oka95a, Oka98]. The basic principle is that we keep a runtime account, where we deposit runtime for fast operations and withdraw runtime for expensive operations. The amortised runtime of an operation is the real runtime plus deposited runtime or minus withdrawn runtime, respectively. Depending on how this account is kept, we distinguish the *physicist's* and the *banker's* method.

The physicist's method In the physicist's method, for each object, a potential is defined in terms of the objects internal structure. The amortised cost is the real-time plus the change of potential (which can of course be negative). In the analysis the worst case for the change of potential is assumed, so that the potential is a lower bound for the actual runtime account.

The banker's method In the banker's method, credits are placed on individual locations in the data structure. The total runtime is the real runtime plus the number of allocated credits minus the number spent credits; of course, only credits that had previously been allocated may be spent. The crucial part of this analysis is the definition of a credit invariant that ensures that expensive operation can spent sufficient credits.

Both these methods unfortunately break down when the analysed data structure is used persistently. Throughout this document, the term *persistence* will not refer to persistent storage accross program invocations, but the possibility to access older versions of data structures even after updates have taken place. While this property is of course inherent to purely functional programming languages, not every program takes advantage of it, justifying the above methods. The problem with these methods under persistent usage is that the same expensive operation can be invoked upon the same argument over and over again, while this would only be possible once without persistency. In terms of the banker's method, this means that credits can be spent more than once, while for the phycisist's method the same potential can be decreased multiple times.

The banker's method and persistency By exploiting lazy evaluation, it is possible to give amortised runtime bounds that are valid even under persistent usage. To achieve this, we will not keep credits from cheap operations, but instead generate debits when we suspend an expensive operation. In particular, on every generated suspension a number of debits corresponding to the runtime needed to evaluate the suspension is placed. Whenever an operation accesses the result of a suspensions, it has to pay of all debits still associated with it. The amortised runtime is the time needed to generate all suspended operations[1] plus the number of discharged debits. The important aspect of the analysis is again to define an invariant on the debits that ensures that any forced suspensions are free of debits.

Thanks to memoization, every generated suspension is evaluated at most once, making this method feasible even in persistent scenarios.

Worst-case bounds Even for data structures relying on lazy evaluation, it is often possible to establish worst-case bounds instead of amortised bounds. The trick is to actually force the suspension when it just would be conceptually paid for in the banker's method. Sometimes this demands some computations to be rewritten to enable incremental instead of monolithic evaluation, but the actual design problem is to find suitable ways to force a computation in a lazy language. This could of course be achieved in Haskell by use of strictness flags, pattern matches and `seq`s. However, if a function that applies one these techniques to force a suspension is called, that call may well be suspended itself, rendering the approach pointless. So an application using a data structure featuring worst-case instead of amortised runtimes, would have to prevent lazy evaluation of all the operations invoked upon the data-structure everywhere. But such an application of Haskell may be considered very unusual, maybe even inappropriate. Therefore, throughout this document, only amortised time bounds will be considered, except for those already implemented data structures that do not rely on amortisation or lazy evaluation.

Benchmarking Some of the data structures described will be strictly better than others when only asymptotic bounds are considered - but quite often, those that are simpler might be faster

[1]Note that we only take suspensions into consideration where we take benefit from it and may assume strict evaluation for other parts to simplify the analysis.

in practice thanks to reduced overhead. Unfortunately, what "in practice" means may vary significantly, depending on the usage scenario. We will therfore not try to distinguish the efficiency of different data structures by use of benchmarks, but limit ourselves to asymptotic bounds. Users that are extremely concerned about performance may either try different implementations in their application, which is easy as they all provide the same interface, or use a tool that is specifically designed to benchmark data structures, namely Auburn [MR98, MR99]. Auburn allows to benchmark data structures using automatically generated "datatype usage graphs", which are parametrised by the relative frequency of the different operations and the degree at which persistency is exploited.

1.2 Current State of Edison

This section is intended to give an overview of the Edison library and its current state of development. As the overall framework is described very well in the manual [Oka99], only the most important aspects of it will be summarised here. Instead, the focus will be laid on the existing implementations, as those are only mentioned briefly or not at all in the manual.

The type classes provided by the framework can be divided into three categories: sequences (like stacks, queues, deques), collections (like sets, bags and priority queues) and associative collections (like maps and priority queues with distinct priorities).

1.2.1 Sequences

Sequences store their elements in the order of insertion, like stacks, queues or deques. There is only one class for sequences Edison: `Sequence`. So all implementations provide the same functionality, differing in which operations are supported efficiently. For example, queues in Edison allow reading and writing both at the front and rear end, but only reading at the front and writing at the rear end will turn out to be efficient.

To become an instance of `Sequence`, a data-type also has to instantiate `Functor` and `MonadPlus` (and hence also `Monad`):

```
class (Functor s, MonadPlus s) ⇒ Sequence s
```

The `fmap` for `Functor` is simply Edison's generalised `map`, while the `MonadPlus` can easily be instantiated with

```
return  = single
xs ≫= k = concatMap k xs
mplus   = append
mzero   = empty
```

Furthermore, for elements of the `Eq` class, the sequence should also be in `Eq`, and likewise for `Show`.

Implementations The sequences section is the one that the most implementations exist for. Furthermore, it is the only section for which the implementations are at least mentioned in the manual. The main concepts of the implementations will be described here, while a detailed analysis of the runtime complexities will be carried out in chapter 2. A more detailed discussion of the underlying principles is left to the cited papers.

`ListSeq` The `ListSeq` is the simplest implementation of the `Sequence` abstraction. It just uses the built-in Haskell lists and makes them an instance of `Sequence` mostly by using the functions from the prelude or with straight-forward implementations of the functionality not found in the prelude. Except for `fromList` the time complexities of `ListSeq` are considered to be default values that the other implementations can be compared with. As `fromList` is trivial for `ListSeq`, $\mathcal{O}(n)$ is used as default value for it.

`SimpleQueue` This is the most common functional implementation of queues as described by Hood and Melville [HM81] (named "BatchedQueue" in [Oka98]). It uses two lists internally, one for the front and one - in reversed order - for the rear end, maintaining the invariant that the front list may only be empty if the rear list is also empty. Although this requires the rear list to be reversed to become the new front list whenever the latter gets empty, amortised time bounds for the standard queue operations (`lhead`, `ltail` and `snoc`) still are $\mathcal{O}(1)$ if used *non-persistently.* If `toList` is used in a setting where the queue is no longer used afterwards (which includes several `Sequence` operations when the queue is used non-persistently), it also has amortised time complexity $\mathcal{O}(1)$.

`BankersQueue` By changing the invariant of the `SimpleQueue` to enforce the front list to be at least as long as the rear list and exploiting lazy evaluation, Okasaki [Oka95c, Oka98] managed to make the queues efficient even when used persistently. However, this comes at the cost of some additional overhead.

Because the `BankersQueue` separately stores the sizes of the two lists, the `size` and `inBounds` operations also are accelerated to $\mathcal{O}(1)$ by this implementation.

`MyersStack` The `MyersStack` [Mye83] is a close relative to the ordinary list, but in addition to the normal tail, every node also maintains a jump-tail which makes it possible to skip some elements when traversing the list. This "some" is determined in a way that makes it possible to reach every position in the list in $\mathcal{O}(\log n)$ time, thereby accelerating the `lookup` and related operations as well as `size` and `rhead`, which essentially just traverse the list to the end.

`RandList` Okasakis `RandList` [Oka95b, Oka98] takes a different approach to allow not only `lookup`, but also `update` (and related operations) in $\mathcal{O}(\log n)$ time. The elements are stored in a list of complete binary trees of increasing size where only the first two trees are allowed to have the same size. Both the length of list and the depth of the largest tree are $\mathcal{O}(\log n)$, thus permitting the demanded time complexity for accessing the ith element, while still preserving $\mathcal{O}(1)$ for `cons`.

`BinaryRandList` The `BinaryRandList` [Oka98] is also a hybrid structure between tree and list. The tail of the list constructor for a list of `a` is no longer a list of `a`, but a list of `(a, a)`. To allow for any number of elements to be stored, an additional constructor exists that takes no head. The resulting structure can also be seen as a complete binary tree where the inner nodes may or may not contain elements, but share this property within one level. While giving $\mathcal{O}(\log n)$ time for `lookup` and `update`, unfortunately, this approach also results in $\mathcal{O}(\log n)$ time for `cons`. However, it should be noted that n consecutive calls to `cons` have a total time-complexity of only $\mathcal{O}(n)$, not $\mathcal{O}(n \log n)$.

`JoinList` The `JoinList` represents the sequence as a binary leaf tree which allows `cons`, `snoc` and `append` in constant time, but slows `lhead` down to $\mathcal{O}(n)$, while `ltail` still runs in $\mathcal{O}(1)$ amortised.

`BraunSeq` As suggested by Hoogerwoord [Hoo92a], Braun trees, i.e. balanced binary trees, can be used to implement flexible arrays where most operations dealing with only one element run in logarithmic time. Okasaki [Oka97b] has further developed implementations for `size` and `copy` that run in $\mathcal{O}(\log^2 n)$ and $\mathcal{O}(\log n)$, respectively, and for `fromList` that runs in $\mathcal{O}(n)$ time.

Adaptors Edison presently contains two adaptors that modify the behaviour of the `Sequence` implementation they are used on.

`Sized` **(in module** `SizedSeq`**)** The `Sized` adaptor stores the size of the underlying `Sequence` explicitly, thereby offering constant-time `size` and `inBounds`, while adding a small overhead to all operations that change the size.

`Rev` **(in module** `RevSeq`**)** The `Rev` uses the underlying `Sequence` in reversed order, i.e. replaces all calls to `cons` by `snoc`, `lhead` by `rhead` etc. and vice versa. Additionally, the size is stored explicitly as for the `Sized` adaptor.

QuickCheck properties For all axioms stated in the user manual, appropriate QuickCheck properties are specified in the supplementary file `TestSeq.hs`. However, the properties are often stated by means of `fromList/toList` constructs instead of direct translations of the axioms. Furthermore, the implementation to be tested is selected by an **import** statement in `TestSeq.hs`, i.e. that file has to be changed whenever a new implementation is to be tested.

1.2.2 Collections

Edison contains a total of eight classes for collections, differing in whether they support uniqueness (e.g. sets), ordering (e.g. priority queues) and observability (see the manual for a detailed discussion of observability).

```
class Eq a ⇒ CollX c a
class (CollX c a, Ord a) ⇒ OrdCollX c a
class CollX c a ⇒ SetX c a
class (OrdCollX c a, SetX c a) ⇒ OrdSetX c a
class CollX c a ⇒ Coll c a
class (Coll c a, OrdCollX c a) ⇒ OrdColl c a
class (Coll c a, SetX c a) ⇒ Set c a
class (OrdColl c a, Set c a) ⇒ OrdSet c a
```

Table 1.1 gives an overview of the collection classes supporting observability; non-observable classes have an `X` appended to their name.

Note that data-types instantiating collection classes do not need to instantiate further classes outside the hierarchy, but contrary to the sequences, the type of the elements is restricted to those in class `Eq` and possibly `Ord`, which makes it necessary for the element type to appear in the class signature.

	unordered	ordered
multi	`Coll`	`OrdColl`
unique	`Set`	`OrdSet`

Table 1.1: Collection classes by supported properties

Implementations Edison currently contains several implementations of heaps (i.e. instances of `OrdColl`) and one implementation of an `OrdSet`. It should be noted that all currently available implementations require the stored elements to be instances of `Ord`.

`UnbalancedSet` The `UnbalancedSet` is the only implementation of a set, namely the `OrdSet` class. It uses a simple, unbalanced binary tree and therefore has a runtime complexity of $\mathcal{O}(n)$ for most operations.

`LeftistHeap` The `LeftistHeap` is an instance of `OrdColl` that uses a heap ordered binary tree maintaining the leftist property that for every node, the right spine of the left child is at least as long as that of the right child. This allows `merge` and therewith many other operations in $\mathcal{O}(\log n)$ [Oka98].

`SplayHeap` This implementation resembles a (binary) splay tree, with the slight difference that in a functional setting, queries cannot, of course, restructure the tree. As a consequence of this, most operations have logarithmic runtime, but `minElem` has time complexity $\mathcal{O}(n)$. Furthermore, if used persistently, the restructuring will have no effect, yielding linear runtimes for most operations. However, when used non-persistently and with caching of the minimum element, the `SplayHeap` is very efficient due to low overheads [Oka98].

`LazyPairingHeap` Pairing heaps where first described in [FSST86] as heap-ordered multiway trees where the linking of the subtrees after a `deleteMin` is carried out in a special way, and constant amortised runtime for the main heap operations was conjectured, but only logarithmic time bounds could be proven. For the *decrease key* operation, which is not part of Edison's framework, Fredman later showed [Fre97] that it is indeed not performed in constant amortised time. For the operations contained in Edison, however, the conjectured runtimes have as of now neither been proven nor disproven. The `LazyPairingHeap` is an adaption of the pairing heap utilising lazy evaluation to maintain the amortised time bounds even under persistent usage.

`SkewHeap` The `SkewHeap` is a functional adaption of the self-adjusting heap described in [ST86]. It is similar to the `LeftistHeap` but instead of maintaining the strict leftist property, it uses a simple restructuring heuristic to reduce the length of the right spine, which is sufficient to achieve the same amortised time bounds.

Adaptors There is only one adaptor available to be used on collection classes: `Min` in the module `MinHeap`. It can be used with instances of `OrdColl` and results in an `OrdColl` that explicitly stores the minimum element. This can be useful in conjuction with data structures that have an expensive `minElem` operation like `SplayHeap`.

QuickCheck properties Two files with QuickCheck properties, one for `OrdColl` and one for `OrdSet` exist. As the latter includes all operations defined for collections, all necessary properties are specified. The Edison manual states no axioms for the collection operations, so these QuickCheck properties can actually be considered normative.

1.2.3 Associative Collections

The classes for associative collections in Edison are very similar to the plain collections, so the eight different classes are distinguished by the same properties, being ordering, uniqueness and observability, with respect to the keys (the elements are, of course, always observable):

```
class Eq k ⇒ AssocX m k
class (AssocX m k, Ord k) ⇒ OrdAssocX m k
class AssocX m k ⇒ FiniteMapX m k
class (OrdAssocX m k, FiniteMapX m k) ⇒ OrdFiniteMapX m k
class AssocX m k ⇒ Assoc m k
class (Assoc m k, OrdAssocX m k) ⇒ OrdAssoc m k
class (Assoc m k, FiniteMapX m k) ⇒ FiniteMap m k
class (OrdAssoc m k, FiniteMap m k) ⇒ OrdFiniteMap m k
```

Table 1.2 gives an overview of the observable association classes; non-observable classes have an `X` appended to their name.

	unordered	ordered
multi	`Assoc`	`OrdAssoc`
unique	`FiniteMap`	`OrdFiniteMap`

Table 1.2: Associative collection classes by supported properties

The same that was said about the elements of the collections also holds for the keys of the associations, while the type of the stored elements is completely unrestricted and hence does not occur in the class signature.

Implementations Currently, Edison contains only two implementations of associative collections, both being instances of `FiniteMap`.

`AssocList` The `AssocList` simply resembles a list with key-element-pairs. Duplicate keys are kept, but only the first occurrence in the list, i.e. the latest insertion is considered valid. While this allows `insert` to be performed in $\mathcal{O}(1)$, it results in the curious property that the size of the data structure is not bounded by the number of (logically) contained bindings. This is especially problematic as many operations take time linear in the physical, not logical size of the `AssocList`.

`PatriciaLoMap` Patricia trees [Mor68, OG98] are derived from binary tries by collapsing paths of common bit-subsequences into a single node. The maximum depth of the tree is still the number of bits of the key, but the average depth is reduced which slightly speeds up `insert` and `lookup`, and more importantly, `union` is significantly accelerated. Edison's `PatriciaLoMap` is fixed for a key type of `Int` and processes the bits of the keys in little-endian order. The type-signatures in the class definitions force the data-type of the `PatriciaLoMap` to take the key type as a parameter; this is however never used and the instances are only defined for `Int` key types.

QuickCheck properties No QuickCheck properties for the associative collections have been implemented, and the Edison manual does not state any axioms. Not even all of the operations are described in the manual, but the intended behaviour can easily be derived from the names and types of the operations and by comparison with the respective collection operations.

1.2.4 Conclusions

Edison provides a nice framework of classes, but as of now, it lacks implementations. Furthermore, the existing implementations are not documented sufficiently. While their functional behaviour is defined by their respective type-classes which are described in the manual, the runtime complexities, being a central point in a library concerned about efficiency, are only documented for a few operations of a few `Sequence` implementations. Even worse, the implementations for the collection and association classes are not even mentioned in the manual. To become useful in practice, the user should be provided with more implementations, especially for those type-classes where none exist so far, and an comprehensive overview of their runtime complexities and other specialities, to make the decision easy which implementation to use.

Unfortunately, reaching this goal in its full breadth is beyond the scope of this project. Instead, we will focus on the `Sequence` part. As that part is the most evolved one so far, it might seem more beneficial to focus on one of the more neglected ones first, providing at least the basics there. However, bringing one part to a point where it might be considered ready for shipping could potentially yield insights useful already in the basic stages of the other parts, so we prefer to do that.

Chapter 2

Runtime Complexity Analysis of Existing Implementations

Unfortunately, the Edison manual only gives runtime complexities for some of the operations of each `Sequence` implementation. In this section, we will analyse those operations that have not before been examined in one of the publications on which the implementations are based.

For some of the operations, the runtime complexity is always the same and more over trivial to see, we therefore exclude these operations from a detailed analysis. They are summarised in table 2.1.

`empty`, `single`, `null`	$\mathcal{O}(1)$
`lview`	maximum of `lhead` and `ltail`
`rview`	maximum of `rhead` and `rtail`
`map`, `foldr`, `foldl`, `foldr1`, `foldl1`, `reducer`, `reducel`, `reduce1`, `tabulate`, `mapWithIndex`, `foldrWithIndex`, `foldlWithIndex`, `filter`, `partition`, `takeWhile`, `dropWhile`, `splitWhile`, `unzipWith`, `unzipWith3`	$n \cdot T_f$, where T_f is the time needed for application of the function argument
`adjust`	same as `update` plus T_f
`lookupM`, `lookupWithDefault`	same as `lookup`
`zip3`	same as `zip`
`zipWith`, `zipWith3`	same as `zip` plus $n_{min} \cdot T_f$
`unzip3`	same as `unzip`

Table 2.1: Common time complexities of all `Sequence` implementations

The given code-fragments are taken from Edison and (C) Chris Okasaki. Some of the functions that are implemented using defaults are replaced by the respective definitions of the defaults. Note that the variables used in the text for function arguments follow appendix A and may differ from the ones used in the code; in general, their meaning should however be apparent from the context.

2.1 `ListSeq`

We take into account that lists are lazy and perform an analysis using the Banker's method with the invariant that the cumulative debt on the first i nodes is less than i, and thus the first node is free of debits. In most cases, the analysis is trivial and we just state the following results:

`cons`, `lhead`, `ltail`, `fromList`: $\mathcal{O}(1)$.

`snoc`, `rhead`, `rtail`, `size`, `reverse`, `unzip`: $\mathcal{O}(n)$.

`append`, `reverseOnto`: $\mathcal{O}(n_1)$.

`concat`: $\mathcal{O}(m+n)$.

`concatMap`: $n \cdot T_f + \mathcal{O}(m)$.

`inBounds`, `lookup`, `update`, `take`, `drop`, `splitAt`: $\mathcal{O}(i)$.

`subseq`: $\mathcal{O}(i+j)$.

`zip`: $\mathcal{O}(n_{min})$.

For all of these, the runtime complexity is in fact the same as if it had been done assuming strictness. Where we can profit from laziness is the `copy` constructor, as it can leave all debits it creates undischarged except for the first one and thus has an amortised runtime of $\mathcal{O}(1)$.

On the other hand, to be consistent with the other implementations, `toList` has to return a list that is free of debits and therefor has to discharge all $\mathcal{O}(n)$ debits, resulting in linear amortised runtime.

2.2 `SimpleQueue`

The `SimpleQueue` is the well-known functional queue implementation using a front and a rear list as proposed in [HM81]. As shown in [Oka98] using both the physicist's and the banker's method, `lhead`, `ltail` and `snoc` run in $\mathcal{O}(1)$ amortised when used non-persistently. We will use the physicist's method to analyse the remaining operations, where the potential is the length of the rear list.

As `cons` only adds an element to the front queue (taking constant time) which does not affect the potential, its amortised runtime is also $\mathcal{O}(1)$.

When two queues are `append`ed, only the potential of the second one is preserved, so the total potential is reduced by the length of the first rear list. This pays of the cost for the `reverseOnto`, leaving the `++`, which runs in time linear in the length of the first front list, i.e. $\mathcal{O}(n_1)$. The same argument, only with `reverseOnto` and `++` exchanged also holds for the queue-`reverseOnto` operation.

For queues with non-empty rear list, both `rhead` and `rtail`, as well as `reverse`, obviously run in $\mathcal{O}(1)$. Otherwise, however, the respective operation on the front list is called, which takes $\mathcal{O}(n)$, while the potential remains unchanged, giving an amortised runtime of $\mathcal{O}(n)$.

Determining the size of a queue needs to determine the length of both lists, and therefore take $\mathcal{O}(n)$ time amortised.

While `fromList` obviously runs in $\mathcal{O}(1)$, `toList` needs $\mathcal{O}(n)$ for the `++` and `reverse` if the rear list is non-empty.

```
data Seq a = Q [a] [a]
  -- invariant: front empty only if rear also empty

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

append (Q xs1 ys1) (Q xs2 ys2) =
    Q (xs1 ++ L.reverseOnto ys1 xs2) ys2

rhead (Q xs (y:ys)) = y
rhead (Q [] []) = error "SimpleQueue.rhead:␣empty␣sequence"
rhead (Q xs []) = L.rhead xs

rtail (Q xs (y:ys)) = Q xs ys
rtail q@(Q [] []) = q
rtail (Q xs []) = Q (L.rtail xs) []

size (Q xs ys) = length xs + length ys

reverse (Q xs []) = Q (L.reverse xs) []
reverse (Q xs ys) = Q ys xs

reverseOnto (Q xs1 ys1) (Q xs2 ys2) =
    Q (ys1 ++ L.reverseOnto xs1 xs2) ys2

fromList xs = Q xs []

toList (Q xs []) = xs
toList (Q xs ys) = xs ++ L.reverse ys

concat = foldr append empty
concatMap f =  foldr (append . f) empty

copy n x = fromList (L.copy n x)

take i s = fromList (L.take i (toList s))
drop i s = fromList (L.drop i (toList s))
splitAt i s = (take i s, drop i s)
subseq i j xs = take j (drop i xs)

lookupM = lookupMUsingDrop
inBounds = inBoundsUsingLookupM
lookup = lookupUsingLookupM

adjust f i xs = fromList (L.adjust f i (toList xs))
update = updateUsingAdjust

zip = xs ys = fromList (L.zip (toList xs) (toList ys))
unzip = unzipUsingLists
```

Listing 2.1: Analysed functions of `SimpleQueue`

Obviously, `concat` requires $\mathcal{O}(n)$ for the `foldr` to traverse the queue, plus $\mathcal{O}(m)$ for all the `append`s to be performed, thus $\mathcal{O}(m+n)$ in total. The same holds for `concatMap`, except that additionally `f` has to be applied `n` times.

As we perform an analysis assuming strictness, we have to completely account for the complete evaluation of the list-`copy` operation, yielding $\mathcal{O}(i)$ runtime for `copy`.

Both `take` and `drop` operate by conversion to a list and back. This reduces the potential to zero, thereby paying of the `reverse`. Furthermore, if the index lies within the front list, for `take` the `++` is only partially evaluated, yielding $\mathcal{O}(i)$ amortised runtime. Unfortunately, this is not true for `drop`, which hence has a runtime complexity of $\mathcal{O}(n)$. This implies of course, that also `splitAt` runs in $\mathcal{O}(n)$.

For `subseq`, things are slightly different, as we only need the first `j` elements of the result of the `drop`, and hence do not necessarily need to completely evaluate the `++`, but only the first `i` steps for the `drop` and `j` steps for the `take`. Furthermore, the `toList` inside the `take` is the trivial case as the rear list is empty, thus does only contribute a constant to the runtime, which is hence $\mathcal{O}(i+j)$.

Likewise, for `lookupM`, only the first element of the `drop` is needed. However, as we do not return a new queue, we do not reduce potential, and hence, the $\mathcal{O}(n)$ complexity of the `reverse` dominates the runtime. This is then of course also the case for `lookup` and `inBounds`, all of which use `lookupM` internally.

All the remaining functions (`update`, `zip` and `unzip`) keep all the elements of the queue in the resulting front list, hence have to evaluate the complete `++` and run in $\mathcal{O}(n)$. (For the `zip`s, the shortest queue is deciding.)

2.3 `BankersQueue`

The above analysis for `SimpleQueue` mostly falls apart under persistent usage: The main point of the amortisation is that before an operations takes $\mathcal{O}(n)$ time because of a rotation, there were $\mathcal{O}(n)$ operations perfoming in constant time as no rotation was needed. But under persistent usage, we can simply invoke the same expensive operation, like an `ltail` on a queue with only one element in the front list, over and over again, as we are allowed to use the original queue even after doing the `ltail`.

By modifying the `SimpleQueue` to exploit lazyness, Okasaki has overcome this problem in the `BankersQueue` and shown that it maintains constant amortised runtime for `lhead`, `ltail` and `snoc` even when used persistently [Oka98]. For the analysis of the remaining functions, we follow Okasakis approach using the banker's method where for the cumulative debits on the first i elements of the front list, $D(i)$, it must hold that $D(i) \leq \min(2i, |f| - |r|)$. (No debits are allowed on the rear list.)

Obviously, `cons` runs in constant time and maintains the debit invariant.

The `++` inside the `append` places one debit on each element of the first $|xs1|$ elements of the resulting front list, while the `reverseOnto` places $|ys1|$ debits on the very next element, in addition to those debits inherited from the original queues. As obviously $|xs|-|ys| \geq |xs1|-|ys1|$ and $|xs| - |ys| \geq |xs2| - |ys2|$, it is sufficient to discharge these n_1 debits to reestablish the debit invariant, yielding amortised runtime $\mathcal{O}(n_1)$. The same can be shown with a very similar argument for `reverseOnto`.

For the analysis of `rhead` and `rtail` we consider the expensive cases where the rear list is empty.

```
data Seq a = Q !Int [a] [a] !Int
  -- invariant: front at least as long as rear

makeQ i xs ys j
  | j > i = Q (i + j) (xs ++ L.reverse ys) [] 0
  | otherwise = Q i xs ys j

cons x (Q i xs ys j) = Q (inc i) (x:xs) ys j

append (Q i1 xs1 ys1 j1) (Q i2 xs2 ys2 j2) =
    Q (i1 + j1 + i2) (xs1 ++ L.reverseOnto ys1 xs2) ys2 j2

reverseOnto (Q i1 xs1 ys1 j1) (Q i2 xs2 ys2 j2) =
    Q (i1 + j1 + i2) (ys1 ++ L.reverseOnto xs1 xs2) ys2 j2

rhead (Q i xs (y:ys) j) = y
rhead (Q _ [] [] _) = error "BankersQueue.rhead:␣empty␣sequence"
rhead (Q i xs [] _) = L.rhead xs

rtail (Q i xs (y:ys) j) = Q i xs ys (dec j)
rtail q@(Q _ [] [] _) = q -- empty case
rtail (Q i xs [] _) = Q (dec i) (L.rtail xs) [] 0

size (Q i xs ys j) = i + j

inBounds s i = i ≥ 0 && i < size s

reverse (Q i xs ys j) = makeQ j ys xs i

fromList xs = Q (length xs) xs [] 0

toList (Q i xs ys j)
  | j == 0 = xs
  | otherwise = xs ++ L.reverse ys

copy n x
  | n < 0     = empty
  | otherwise = Q n (L.copy n x) [] 0

lookup (Q i xs ys j) idx
  | idx < i   = L.lookup xs idx
  | otherwise = L.lookup ys (j - (idx - i) - 1)
```

Listing 2.2: Analysed functions of BankersQueue, part 1

In this case, the respective list-function, taking $\mathcal{O}(n)$ has to be called. As the result of the list-`rtail` is monolithic, all of its debits have to be discharged immediately, giving linear amortised runtime for both operations.

Thanks to the explicitly stored lengths of the lists, `size` and `inBounds` obviously run in $\mathcal{O}(1)$.

The `reverse` operation runs in $\mathcal{O}(n)$, which is easily seen by considering the worst-case where the rear-list is empty. Then, the front-list has to be reverse and all of the thereby created debits have to be discharged immediately.

As conversion from a list requires the length of that list to be determined, `fromList` has time complexity $\mathcal{O}(n)$. The `++` and `reverse` for conversion to a list also need $\mathcal{O}(n)$ in total, in addition to the $\mathcal{O}(n)$ debits that might have to be discharged.

Contrary to what was the case for `SimpleQueue`, the `copy` constructor can leave the debits on the front list, and therefore runs in $\mathcal{O}(1)$ amortised.

The `lookup`-family of operations, as well as `update` directly call the corresponding list-function on the respective list. If the index lies in the front queue, this obviously results in a time complexity of $\mathcal{O}(i)$. Otherwise, it is $\mathcal{O}(|r|)$, and so by $|r| \leq |f| < i$, also $\mathcal{O}(i)$.

If the index to `take` lies in the front list, the call to the list-`take` takes i steps. Additionally, it might be necessary to discharge i debits, as there may have been a total of $2i$ debits on the first i elements of the front list, but the total number is afterwards limited by $|f| - |r| = i$. If the index lies in the rear list, the drop obviously runs in $\mathcal{O}(|r|)$, and hence $\mathcal{O}(i)$, and preserves the debit invariant.

For `drop`, we distinguish three cases: the index lies in the front list, rotation is not needed; the index lies in the front list, rotation is needed; or the index lies in the rear list. In the first case, the list-`drop` requires i steps, and we have to discharge not only the debits on the first i elements, but also additional i debits on the resulting front list because of the index-shift. If a rotation is needed, we know that (before the `drop`) $|f| - i < |r|$, hence the number of debits on the front list is bounded by i. We discharge these and perform the list-`drop` in $\mathcal{O}(i)$. The rotation then places one debit on the first $|f| - i$ nodes of the new front list each and $|r|$ on the very next one. To reestablish $D(i) \leq 2i$, we discharge the debit on the first node. We then have $D(|f| - i) = |f| - i - 1 + |r|$, so it is obviously sufficient to discharge further $i - 1$ debits from the $|f| - i$th node to reestablish the debit invariant. Finally, if the index lies in the rear-list, the list-`take` and `reverse` both run in Oof—r—-i and require all their debits to be discharged immediately, and as $|r| \leq |f| < i$, we therewith arrive at an amortised runtime complexity of $\mathcal{O}(i)$.

Combining the arguments for `take` and `drop`, it is easy to see that `splitAt` is also $\mathcal{O}(i)$ amortised. It also follows directly that `subseq` runs in $\mathcal{O}(i + j)$.

As `concat` directly relies on `append`, it obviously runs in $\mathcal{O}(m+n)$. Similarly, `concatMap` requires $\mathcal{O}(m)$ to perform all the `appends`, in addition to the time required applying `f` n times.

The `zip`-family of operations uses the list representation internally. However it should be noted that only the first n_{min} (length of the shortest queue) elements of each queue are actually accessed, so that the `reverse` of the `toList` cannot be forced for queues longer than $2n_{min}$. Hence we can establish an amortised runtime complexity of $\mathcal{O}(n_{min})$. The `unzip` operations trivially run in $\mathcal{O}(n)$.

```
update idx e q@(Q i xs ys j)
  | idx < i = if idx < 0 then q
              else Q i (L.update idx e xs) ys j
  | otherwise = let k' = j - (idx - i) - 1
                in if k' < 0 then q
                   else Q i xs (L.update k' e ys) j

take len q@(Q i xs ys j) =
  if len ≤ i then
    if len ≤ 0 then empty
    else Q len (L.take len xs) [] 0
  else let len' = len - i in
    if len' ≥ j then q
    else Q i xs (L.drop (j - len') ys) len'

drop len q@(Q i xs ys j) =
  if len ≤ i then
    if len ≤ 0 then q
    else makeQ (i - len) (L.drop len xs) ys j
  else let len' = len - i in
    if len' ≥ j then empty
    else Q (j - len') (L.reverse (L.take (j - len') ys)) [] 0

splitAt idx q@(Q i xs ys j) =
  if idx ≤ i then
    if idx ≤ 0 then (empty, q)
    else let (xs',xs'') = L.splitAt idx xs
         in (Q idx xs' [] 0, makeQ (i - idx) xs'' ys j)
  else let idx' = idx - i in
    if idx' ≥ j then (q, empty)
    else let (ys', ys'') = L.splitAt (j - idx') ys
         in (Q i xs ys'' idx', Q (j - idx') (L.reverse ys') [] 0)

subseq i len xs = take len (drop i xs)

concat = foldr append empty
concatMap  f = foldr (append . f) empty

zip xs ys = fromList (L.zip (toList xs) (toList ys))
unzip = unzipUsingLists
```
Listing 2.3: Analysed functions of `BankersQueue`, part 2

2.4 MyersStack

The `MyersStack` is very similar to an ordinary list, but in addition to the normal tail, every node also contains an additional jump-tail, that points some elements further down the list, and the number of elements that can be skipped by following the jump-tail. As shown by Myers [Mye83], if the jump-tails are constructed in the right way, it is possible to reach any element in $\mathcal{O}(\log n)$ steps, while most of the other operations inherit their runtime complexity from the normal list. In particular, Myers considers `cons`, `ltail` (runtime $\mathcal{O}(1)$) and `lookup` (runtime $\mathcal{O}(\log n)$), as well as `size`. However, he uses a slightly different representation that allows for `size` to be determined in constant time, which is not possible for this implementation. The argument for `lookup` can easily be adopted for `inBounds` and `drop`. It is furthermore trivial to see that all these operations are also bounded by $\mathcal{O}(i)$, so that we can give the slightly better bound $\mathcal{O}(\min(i, \log n))$.

We will not discuss those operations here that are essentially the same as for normal lists, only adding a constant overhead to maintain the jump-tails, leaving only a few that deserve analysis.

```
data Seq a = E | C !Int a (Seq a) (Seq a)

jump (C _ _ _ (C _ _ _ xs')) = xs'

size xs = go xs
  where go E = (0::Int)
        go (C j x xs xs') = j + size xs'

update i y xs = upd i xs
  where upd i E = E
        upd 0 (C j x xs xs') = C j y xs xs'
        upd i (C j x xs _)
            | j == 1    = C j x ys ys
            | otherwise = C j x ys (jump ys)
          where ys = upd (i - 1) xs

fromList = L.foldr cons empty
toList = foldr (:) []
copy n x = fromList (L.copy n x)
subseq i len xs = take len (drop i xs)
```

Listing 2.4: Analysed functions of `MyersStack`

As `size` recurses over the jump-tail, which is the shortest path to the end of the list, it also runs in $\mathcal{O}(\log n)$.

Unfortunately, the argument for `lookup` does not hold for `update`, as they have to rebuild the list up to the ith element, which takes $\mathcal{O}(i)$ steps.

Conversion to and from a list is performed by applying `foldr` with `cons`, yielding linear runtime. Note also that the list returned by `toList` can be evaluated incrementally. Then obviously `copy` takes $\mathcal{O}(i)$.

Finally, `subseq` again is a combination of `take` and `drop` and therefore runs in $\mathcal{O}(\min(i, \log n)+j)$.

```
data Tree a = L a | T a (Tree a) (Tree a)   deriving (Eq)
data Seq a = E | C !Int (Tree a) (Seq a)    --deriving (Eq)

copy n x = if n ≤ 0 then E else buildTrees (1::Int) (L x)
  where buildTrees j t
          | j > n     = takeTrees n (half j) (child t) E
          | otherwise = buildTrees (1 + j + j) (T x t t)

        takeTrees i j t xs
          | i ≥ j = takeTrees (i - j) j t (C j t xs)
          | i > 0  = takeTrees i (half j) (child t) xs
          | otherwise = xs

        child (T x s t) = t

rhead E = error "RandList.rhead:␣empty␣sequence"
rhead (C _ t E) = treeLast t
  where treeLast (L x) = x
        treeLast (T x s t) = treeLast t
rhead (C _ t xs) = rhead xs

rtail = rtailUsingLview
```
Listing 2.5: Analysed functions of `RandList`, part 1

2.5 `RandList`

As explained by Okasaki [Oka95b, Oka98], the `RandList` (or Skew Binary Random-Access List) preserves constant time bounds for `cons`, `lhead` and `ltail`, while giving a complexity for the `lookup` and `update` operations of $\mathcal{O}(\min(i, \log n))$. The `RandList` resembles a list of pre-ordered complete binary trees, strictly increasing in size, except for the first two that may be of the same size. As the data structure does not rely on lazy-evaluation or amortisation, we will analyse the remaining function as if evaluation was strict.

The `copy` constructor works in two phases. First, the smallest complete tree containing more than n elements (i.e. $2^i - 1$, $i = \lceil \log(n+1) \rceil + 1$) is constructed. As both children of each node are the same, each level is constructed in constant time, and so the overall time is logarithmic. Second, of this tree, subtrees are selected and put into sequence such that the total number of elements gets correct. As at most one tree from every level is taken (with the exception that the smallest taken tree can be taken twice), this is obviously also performed in logarithmic time, and so the total runtime complexity of `copy` is $\mathcal{O}(\log n)$.

As `lookup` runs in $\mathcal{O}(\log n)$, of course so does `rhead`. For `rtail`, on the other hand, the whole sequence is traversed and then reconstructed omitting the last element, requiring linear time.

To determine the size of a `RandList`, it is sufficient to add the sizes of the trees, explicitly stored in the list. As the length of the list is bounded by $\mathcal{O}(\log n)$, so is the time of `size`.

`reverseOnto` traverses the complete sequence that is to be reversed and adds the elements to the second sequence one by one, and as `cons` runs in $\mathcal{O}(1)$, therefore has linear runtime. From this, obviously also `reverse` runs in $\mathcal{O}(n)$.

Due to the tree sizes stored in the list, `inBounds` does not have to traverse down the tree to determine whether an index lies inside it. And as the length of the list up to the tree including index i is bounded by $\mathcal{O}(\log i)$, so is the runtime if `inBounds`.

Proceeding similar to a lookup, `drop` first traverses down the list, dropping the unneeded trees, and then traverses down the tree in which the index lies. In each step, the sub-function `drpTree` descends at least one level and skips over at least one element in constant time, so that the total runtime again is bounded by $\mathcal{O}(\min(i, \log n))$, like for `lookup`.

Both `snoc` and `append` use `foldr` to traverse the sequence that is to be the front part of the result and successively `cons` the respective elements. As `cons` runs in $\mathcal{O}(1)$, the total runtime thus becomes linear.

The same holds for `fromList` and `toList`. It should be noted however, that laziness allows the list generated by `toList` to be built incrementally.

`concat` and `concatMap` are another incarnation of the `foldr` scheme, using `append` instead of `cons`. All these `append`s take time linear in the length of the result, while the the overhead `foldr` is linear in the length of the input, yielding the usual $\mathcal{O}(n + m)$ runtime (plus the time to apply `f` for `concatMap`).

Observing that `toList` is incremental, it follows directly that `take` runs in $\mathcal{O}(i)$, and so does `splitAt`. Furthermore, `subseq` runs in $\mathcal{O}(\min(i, \log n) + j)$, the sum of the runtimes of `take` and `drop`.

Similarly, the `zips` run in $\mathcal{O}(n_{min})$ and the `unzips` in $\mathcal{O}(n)$.

```
size xs = sz xs
  where sz E = (0::Int)
        sz (C j t xs) = j + sz xs

reverseOnto E ys = ys
reverseOnto (C _ t xs) ys = reverseOnto xs (revTree t ys)
  where revTree (L x) ys = cons x ys
        revTree (T x s t) ys = revTree t (revTree s (cons x ys))

reverse s = reverseOnto s empty

inBounds xs i = inb xs i
  where inb E i = False
        inb (C j t xs) i
          | i < j     = (i ≥ 0)
          | otherwise = inb xs (i - j)

drop n xs = if n < 0 then xs else drp n xs
  where drp i E = E
        drp i (C j t xs)
            | i < j     = drpTree i j t xs
            | otherwise = drp (i - j) xs

        drpTree 0 j t xs = C j t xs
        drpTree i j (L x) xs = error "RandList.drop:␣bug.␣␣Impossible␣case!"
        drpTree i j (T x s t) xs
            | i > k     = drpTree (i - 1 - k) k t xs
            | otherwise = drpTree (i - 1) k s (C k t xs)
          where k = half j

snoc s x = foldr cons (single x) s
append s t | null t = s
           | otherwise = foldr cons t s

fromList = L.foldr cons empty
toList = foldr (:) []

concat = foldr append empty
concatMap f = foldr (append . f) empty

take i s = fromList (L.take i (toList s))
split i s = (take i s, drop i s)
subseq i len xs = take len (drop i xs)

zip xs ys = fromList (L.zip (toList xs) (toList ys))
unzip = unzipUsingLists
```

Listing 2.6: Analysed functions of `RandList`, part 2

2.6 BinaryRandList

The BinaryRandList is a hybrid data structure between list and tree. The constructors are similar to that of a list, except that the tail is not a Seq a, but a Seq (a, a), which results in a binary tree-like shape. To be able to store any number of elements, inner nodes do not necessarily contain elements; or, using the list-perspective, there is also a constructor that has only a tail, but no head. Yet another perspective is to see the head elements in the Odd constructor as complete binary trees, represented by nested pairs, which makes it quite similar to the RandList.

As argued by Okasaki [Oka98], this gives logarithmic runtimes for cons, lhead and ltail, but also for lookup. As again neither laziness nor amortisation is used, we can restrict the runtime analysis to a strict worst-case analysis.

The worst case for append is obviously the one that relies on the foldr. It needs $\mathcal{O}(n_1)$ just for the foldr to traverse the first sequence, plus the time needed for all the cons. As mentioned above, cons takes logarithmic time in the worst case; however, the average of successive calls to cons is better. Actually, half the calls to cons only take one step, a quarter only two, and so on, i.e. the average time for cons converges to $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$. But as even for small n_1 the logarithmic case of cons may be triggered, we have to take it into account and arrive at a total runtime complexity of $\mathcal{O}(n_1 + \log n_2)$ for append. The same holds of course also for reverseOnto. From this follows then directly that reverse runs in $\mathcal{O}(n)$.

Both rhead and size recursively traverse the list, the length of which is logarithmic in the number of elements, and hence, so is their runtime complexity.

Unfortunately, rtail has to construct its result element by element, and therefore has linear runtime.

Similar to the RandList, the copy constructor can take advantage of the fact that both subtrees in every level are the same to achieve $\mathcal{O}(\log n)$ runtime.

Observing that inBounds halves its argument in every recursive call, it is easy to see that it runs in $\mathcal{O}(\log i)$.

As for the RandList, conversion from and to a list is performed using foldr. With the same argument as for append, the successive cons invocations during fromList have constant average time, and as we start with an empty BinaryRandList, we thus get linear runtime in total. It should also be noted that toList can once again be evaluated incrementally. However, it may take up to $\mathcal{O}(\log n)$ steps until the first element is determined, not the usual $\mathcal{O}(1)$.

Observing this, it is easy to see that the worst case of take, the call to takeUsingLists, runs in $\mathcal{O}(i + \log n)$.

The analysis of drop is one of the trickier ones. It is easy to see that the maximum recursion depth is $\log i$, but in every step, ltail may be called, requiring up to $\log n$ steps itself. We will show however, that drop still runs in $\mathcal{O}(\log n)$. In order to see this, we first consider what the result looks like after performing all the recursive drps, but none of the ltails. It consists of $\log i$ Even constructors, possibly intermitted with ltails, e.g. something like

```
ltail (Even (Even (Even (ltail (Even (ltail (Even rest)))))))
```

where rest is the remaining part after the recursion terminates (assuming $i < n$). Observe that an arbitrary number of successive Evens is possible, but between two ltails, there has to be at least one Even. Now performing the first step of each ltail results in[1]

[1]For the sake of brevity, we ignore the elements stored in the Odd constructors and the fact that ltail does not call itself recursively, but rather lview.

```
data Seq a = E | Even (Seq (a,a)) | Odd a (Seq (a,a))    deriving (Eq)

mkEven E = E
mkEven ps = Even ps

append xs E = xs
append xs ys@(Even pys) =
  case xs of
    E → ys
    Even pxs → Even (append pxs pys)
    Odd x pxs → Odd x (append pxs pys)
append xs ys@(Odd _ _) = foldr cons ys xs

reverseOnto xs ys = foldl (flip cons) ys xs

reverse s = reverseOnto s empty

rhead E = error "BinaryRandList.rhead:␣empty␣sequence"
rhead (Even ps) = snd (rhead ps)
rhead (Odd x E) = x
rhead (Odd x ps) = snd (rhead ps)

rtail = rtailUsingLview

size E = 0
size (Even ps) = 2 * size ps
size (Odd x ps) = 1 + 2 * size ps

copy n x
    | n ≤ 0 = E
    | otherwise = cp n x
  where cp :: Int → a → Seq a
        cp n x
          | odd n = Odd x (cp (half n) (x,x))
          | n == 0 = E
          | otherwise = Even (cp (half n) (x,x))

inBounds xs i = (i ≥ 0) && inb xs i
  where inb :: Seq a → Int → Bool
        inb E i = False
        inb (Even ps) i = inb ps (half i)
        inb (Odd x ps) i = (i == 0) || inb ps (half (i-1))

fromList = L.foldr cons empty
toList = foldr (:) []
```

Listing 2.7: Analysed functions of `BinaryRandList`, part 1

```
take n xs = if n ≤ 0 then E else tak n xs
  where tak :: Int → Seq a → Seq a
        tak 0 xs = E
        tak i E = E
        tak i (Even ps)
          | even i = Even (tak (half i) ps)
        tak i (Odd x ps)
          | odd i = Odd x (tak (half (i-1)) ps)
        tak i xs = takeUsingLists i xs

drop n xs = if n ≤ 0 then xs else drp n xs
  where drp :: Int → Seq a → Seq a
        drp 0 xs = xs
        drp i E = E
        drp i (Even ps)
          | even i = mkEven (drp (half i) ps)
          | otherwise = ltail (mkEven (drp (half i) ps))
        drp i (Odd _ ps)
          | odd i = mkEven (drp (half (i-1)) ps)
          | otherwise = ltail (mkEven (drp (half (i-1)) ps))

splitAt i s = (take i s, drop i s)
subseq i len xs = take len (drop i xs)

snoc s x = foldr cons (single x) s
concat = foldr append empty
concatMap f = foldr (append . f) empty

zip xs ys = fromList (L.zip (toList xs) (toList ys))
unzip = unzipUsingLists
```

Listing 2.8: Analysed functions of `BinaryRandList`, part 2

```
Odd (ltail (Even (Even (Odd (ltail (Odd (ltail rest)))))))
```

Observe that every `ltail` produces an `Odd` constructor, at which the recursion of the next-outer `ltail` will stop. Thus, the total runtime of all but the inner-most `ltail` is bounded by $\mathcal{O}(\log i)$. And as the length of the rest is $\mathcal{O}(\log n - \log i)$, so is the complexity of the inner-most `ltail`, giving a total runtime of $\mathcal{O}(\log n)$ for all the `ltail`s, plus $\mathcal{O}(\log i)$=$\mathcal{O}(\log n)$ for the recursion of `drp`.

Considering the runtimes of `take` and `drop`, it is now easy to see that `splitAt` runs in $\mathcal{O}(i+\log n)$ and `subseq` in $\mathcal{O}(j + \log n)$.

Taking into account once again that the average time for successive `cons` is constant, we can easily establish linear runtime for `snoc`. But more importantly, as `concat` starts from an empty sequence, the long chains of successive `cons` operations generated by it do not have a logarithmic component in their total runtime like it was the case for `append`, but are just linear in the length of the output. Taking into account the time for the `foldr` itself, which is linear in the input length, this again gives the usual $\mathcal{O}(m+n)$ for `concat`.

Taking into consideration the runtime of `toList`, and especially that the evaluation of even only the first element may take $\mathcal{O}(\log n)$, the `zip` operations obviously take $\mathcal{O}(n_{min} + \log n_{max})$. The `unzip` operations run in $\mathcal{O}(n)$ as usual.

2.7 JoinList

The `JoinList` uses a binary leaf tree to represent a sequence. Unfortunately, Okasaki gives no literature reference for it where at least the runtime of the basic operations would be analysed. The only hint is the Edison manual, that states constant runtime for `snoc` and `append` and linear runtime for `lview`, `ltail`, `rview` and `rtail`, "but $\mathcal{O}(1)$ in practice." By using the Physicists's method for non-persistent usage, we will perform a detailed analysis and establish that `ltail` in fact runs in $\mathcal{O}(1)$ amortised. The potential we use is defined as follows:

$$\begin{array}{rcll} \Phi(\langle x\rangle) & = & 0 & \text{for a leaf node} \\ \Phi(\langle l,r\rangle) & = & \max\left(\Phi(l)+1, \Phi(r)-1\right) & \text{for an inner node with children } l \text{ and } r \end{array}$$

Thus, $\Phi(t)$ is the largest $\phi(p_i)$, where p_i is a path in t from the root to a leaf and $\phi(p)$ is the number of left branches minus the number of right branches taken in p.

All `cons`, `snoc` and `append` clearly run in $\mathcal{O}(1)$. As the former two are merely special cases of the latter, we focus on the change of potential for `append`. Before joining two trees t_1 and t_2, the total potential obviously is $\Phi(t_1)+\Phi(t_2)$ before the join. So the change is $\max\left(\Phi(t_1)+1, \Phi(t_2)-1\right) - \Phi(t_1) - \Phi(t_2) \leq 1$, yielding the desired constant amortised runtime.

As `lhead` has to traverse the whole left spine of the tree, which can be of length n, and does not change the potential, it needs runtime $\mathcal{O}(n)$. But while `ltail` also has to traverse the whole left spine it also restructures the tree; in particular, it performs as many right-rotations on the root as possible, i.e. until the left spine is down to length one. To determine the change in potential

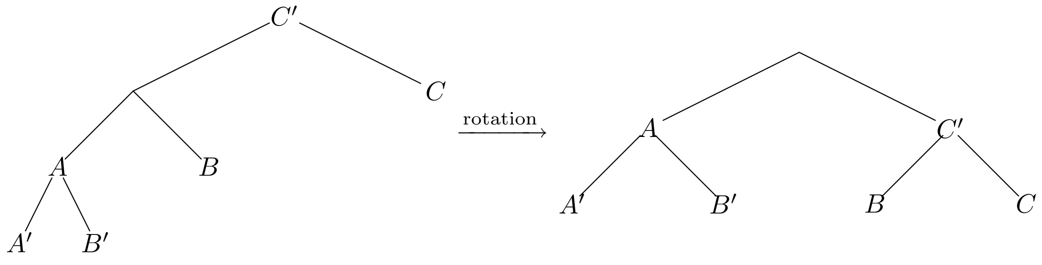


Figure 2.1: A right-rotation

this causes, we look at one single rotation and what the consequences for further rotations are. Figure 2.1 depicts a right rotation. The potential before the rotation is given by

$$\Phi = \max\left(\max\left(\Phi(A)+1, \Phi(B)-1\right)+1, \Phi(C)-1\right) = \max\left(\Phi(A)+2, \Phi(B), \Phi(C)-1\right),$$

while the potential afterwards is

$$\Phi' = \max\left(\Phi(A)+1, \max\left(\Phi(B)+1, \Phi(C)-1\right)-1\right) = \max\left(\Phi(A)+1, \Phi(B), \Phi(C)-2\right).$$

```
data Seq a = E | L a | A (Seq a) (Seq a)

cons x E = L x
cons x xs = A (L x) xs

snoc E x = L x
snoc xs x = A xs (L x)

append E ys = ys
append xs E = xs
append xs ys = A xs ys

lhead E = error "JoinList.lhead:␣empty␣sequence"
lhead (L x) = x
lhead (A xs ys) = lhead xs

ltail E = E
ltail (L x) = E
ltail (A xs ys) = ltl xs ys
  where ltl E zs = error "JoinList.ltl:␣bug"
        ltl (L x) zs = zs
        ltl (A xs ys) zs = ltl xs (A ys zs)

rhead E = error "JoinList.rhead:␣empty␣sequence"
rhead (L x) = x
rhead (A xs ys) = rhead ys

rtail E = E
rtail (L x) = E
rtail (A xs ys) = rtl xs ys
  where rtl xs (A ys (A zs s)) = A (A xs ys) (rtl zs s)
        rtl xs (A ys (L _)) = A xs ys
        rtl xs (L x) = xs

size xs = sz xs (0::Int)
  where sz E n = n
        sz (L x) n = n + (1::Int)
        sz (A xs ys) n = sz xs (sz ys n)

reverse (A xs ys) = A (reverse ys) (reverse xs)
reverse xs = xs -- L x or E

reverseOnto = append . reverse
```

Listing 2.9: Analysed functions of `JoinList`, part 1

We will now differentiate between three different cases, depending on which of the terms in the original potential constituted the maximum.

Case A, $\Phi(A) + 2 > \Phi(B), \Phi(A) + 2 \geq \Phi(C) - 1$: Obviously, $\Phi' = \Phi - 1$, i.e. the potential is decreased by one.

Case B, $\Phi(B) \geq \max(\Phi(A) + 2, \Phi(C) - 1)$: As $\Phi = \Phi(B) = \Phi'$, the potential is not reduced in this case. But since

$$\begin{aligned}\Phi(C') - 1 &= \max(\Phi(B) + 1, \Phi(C) - 1) - 1 \\ &= \Phi(B) \\ &\geq \Phi(A) + 2 \\ &= \max(\Phi(A') + 3, \Phi(B') + 1) \\ &> \max(\Phi(A') + 2, \Phi(B')),\end{aligned}$$

we know that the next rotation will be of case C.

Case C, $\Phi(C) - 1 > \max(\Phi(A) + 2, \Phi(B))$: Like in case A, it is easy to see that the potential is reduced by one. Furthermore, we can again see that

$$\begin{aligned}\Phi(C') - 1 &= \max(\Phi(B) + 1, \Phi(C) - 1) - 1 \\ &\geq \Phi(C) - 2 \\ &> \Phi(A) + 1 \\ &= \max(\Phi(A') + 2, \Phi(B')),\end{aligned}$$

and so the next, and hence all consecutive rotations are of case C.

The important insight that at most once during one call to `ltail`, case B can occur, hence for at most one rotation, no decrease of potential takes place. And, as the final removal of the head element and the caused increase of potential are bounded by $\mathcal{O}(1)$, we indeed get constant amortised runtime for `ltail`. Not that as an exception to the rule, `lview` runs in $\mathcal{O}(1)$ rather than in $\mathcal{O}(n)$, as it also can take advantage of amortisation, while `lhead` cannot.

While `rhead` directly corresponds to `lhead`, except that it traverses the right spine, and therefore also runs in $\mathcal{O}(n)$, `rtail` is substantially different from `ltail`. It also performs rotations, but only one per level, as the recursive call is not performed on the new root, but on the new right child. Every left-rotation increases the potential by at most one, and for every second node on the right spine a rotation is performed, hence the total runtime and increase in potential is $\mathcal{O}(n)$.

The `size` operation has to traverse the whole tree, hence runs in $\mathcal{O}(n)$. The same holds for `reverse`, but we also have to consider the change of potential. It is easy to see that the worst case occurs for a tree in which all inner nodes are on the right spine ($\Phi = 1$), which after the reversal only consists of a left spine ($\Phi = n - 1$). So the increase of potential is $\mathcal{O}(n)$ as well. Considering this, obviously `reverseOnto` runs in $\mathcal{O}(n_1)$.

For `toList`, the tree is traversed essentially in the same way as for `size`, resulting in amortised runtime $\mathcal{O}(n)$ again. Unfortunately, reaching the first element can already take $\mathcal{O}(n)$, so that we cannot exploit incrementality.

As for the other data structures working on trees, th `copy` operation run in $\mathcal{O}(\log n)$ as the argument is halved in every recursive call. It is also easy to see that the potential of the resulting tree is $\mathcal{O}(\log n)$, yielding logarithmic amortised runtime.

Both `concat` and `concatMap` use `foldr`, which results in the n-fold application of `append`. As `append` runs in $\mathcal{O}(1)$ amortised, `concat` has amortised runtime $\mathcal{O}(n)$, while for `concatMap`, invocation of `f` has to be taken into account as well, which will dominate the total runtime.

As `cons` runs in $\mathcal{O}(1)$, `fromList` obviously runs in $\mathcal{O}(n)$.

Both `drop` and `take` run in $\mathcal{O}(i)$ amortised, as the `ltail`/`lview` and `cons` used internally run in $\mathcal{O}(1)$. For `splitAt`, we have to be a bit more careful, as it return two new sequences and we have to consider the sum of both potentials. However, the front part is constructed by series of `cons` and hence has potential one, so we can maintain amortised runtime $\mathcal{O}(i)$ also for `splitAt`. For `subseq`, amortised runtime $\mathcal{O}(i+j)$ follows directly from `take` and `drop`.

All the `lookup` operations and `inBounds` use `drop` to (try to) access the demanded element. However, as they do not result in a new `JoinList`, we cannot simply rely on the amortised cost of `drop`. On the contrary, even accessing only the first element can take $\mathcal{O}(n)$. We can use the amortisation analysis above however to ensure that the total runtime for access to the ith element is also bounded by $\mathcal{O}(n)$.

On the other hand, `update` does return a new sequence, and by adding the amortised runtimes for `splitAt` and `append`, we easily arrive at amortised runtime $\mathcal{O}(i)$.

The `zip` family of operations uses `lview` and `cons` to decompose the inputs and construct the output. Both run in $\mathcal{O}(1)$ amortised and are applied $\mathcal{O}(n_{min})$ times, which hence is the total amortised runtime. Similarly, the `unzip`s use `foldr` to traverse the input and `cons` to construct the output, for an amortised runtime of $\mathcal{O}(n)$.

```
toList xs = tol xs []
  where tol E rest = rest
        tol (L x) rest = x:rest
        tol (A xs ys) rest = tol xs (tol ys rest)

copy n x
    | n ≤ 0 = E
    | otherwise = cpy n x
  where cpy n x  -- n > 0
          | even n = let xs = cpy (half n) x
                     in A xs xs
          | n == 1 = L x
          | otherwise = let xs = cpy (half n) x
                        in A (L x) (A xs xs)

concat = foldr append empty
concatMap f = foldr (append . f) empty

fromList = L.foldr cons empty

take = takeUsingLview
drop = dropUsingLtail
splitAt = splitAtUsingLview
subseq i len xs = take len (drop i xs)

inBounds = inBoundsUsingDrop
lookup = lookupUsingDrop

update = updateUsingSplitAt

zip = zipUsingLview

unzip = unzipUsingFoldr
```

Listing 2.10: Analysed functions of `JoinList`, part 2

2.8 BraunSeq

The `BraunSeq` uses Braun trees, a very strictly balanced variant of binary trees to represent sequences as proposed by Hoogerwoord [Hoo92a]. He has shown that `cons`, `ltail`, `lookup` and `update` all run in $\mathcal{O}(\log n)$ worst-case. As the shape of the tree for the first i elements is independent of the total number of elements n, it is furthermore easy to see that for `lookup` and `update`, the better bound $\mathcal{O}(\log i)$ can be given. The same holds for `inBounds`, which proceeds in the same way as `lookup` to find the requested element, if possible. The runtime complexities for `snoc` given in [Hoo92a] assume explicitly stored sizes in each node, which this implementation does not feature.

Instead, a `size` operation that runs in $\mathcal{O}(\log^2 n)$ is used as derived by Okasaki in [Oka97b], where furthermore an implementation for `copy` running in $\mathcal{O}(\log i)$ and a `fromList` with linear runtime are given. He also suggests inverting the latter to obtain `toList` with linear runtime, which is exactly the way it is implemented here.

As mentioned above, the $\mathcal{O}(\log n)$ bound for `snoc` given in [Hoo92a] assumes known sizes. However, with this implementation, the size has to be determined beforehand, taking $\mathcal{O}(\log^2 n)$, and then the appropriate position in the tree can be reached in $\mathcal{O}(\log n)$. So the `size` operation dominates the runtime of `snoc` to be $\mathcal{O}(\log^2 n)$. The `rtail` operation proceeds in pretty much the same way, hence also requires $\mathcal{O}(\log^2 n)$.

Similarly, `rhead` uses `size` to determine the index of the last element and then calls `lookup`, so that again `size` determines the overall runtime.

The `append` operations recurses over the left sequence in a way that it calls itself twice, but with half the length for the left sequence. This obviously amounts to a total of $\mathcal{O}(n_1)$ recursive invocations. In each step however, the `cons` takes time $\mathcal{O}(\log n_2)$, yielding total runtime of $\mathcal{O}(n_1 \log n_2)$. From this, it follows directly that `concat` runs in

$$\sum_{i=1}^{n} \max\left(1, m_i \log\left(\sum_{j=i+1}^{n} m_j\right)\right) \leq \sum_{i=1}^{n} \max\left(1, m_i \log m\right) = \mathcal{O}(n + m \log m).$$

Like `append`, `take` also calls itself twice with a halved argument, but as it does not call any other expensive operation, it thus has a runtime complexity of $\mathcal{O}(i)$. On the other hand, `drop` does call `combine` in every step which takes $\mathcal{O}(\log n)^2$ time, and thus requires $\mathcal{O}(i \log n)$ time in total. From this, it is easily seen that `splitAt` and `subseq` run in $\mathcal{O}(i \log n)$ and $\mathcal{O}(j + i \log n)$, respectively.

While the `reverse` operation looks quite intimidating, determining its runtime complexity poses no problems, as obviously the four sub-functions again call each other two times with halved arguments, starting from the total size of the sequence, so that the total runtime is linear. Combining `reverse` and `append` then yields $\mathcal{O}(n_1 \log n_2)$ runtime complexity for `reverseOnto`.

All the `zip` operations recurse over the trees until the shortest one has been completed, hence run in $\mathcal{O}(n_{min})$. The `unzip`s traverse the whole tree in a similar fashion and also have linear runtime.

[2] `combine` is an integral part of `ltail` which is analysed in [Hoo92a].

```
data Seq a = E | B a (Seq a) (Seq a)    deriving (Eq)

snoc ys y = insAt (size ys) ys
  where insAt 0 _ = single y
        insAt i (B x a b)
          | odd i     = B x (insAt (half i) a) b
          | otherwise = B x a (insAt (half i - 1) b)

-- not exported
delAt 0 _ = E
delAt i (B x a b)
  | odd i     = B x (delAt (half i) a) b
  | otherwise = B x a (delAt (half i - 1) b)

rtail E = E
rtail xs = delAt (size xs - 1) xs

rhead E = error "BraunSeq.rhead:␣empty␣sequence"
rhead xs = lookup xs (size xs - 1)

append xs E = xs
append xs ys = app (size xs) xs ys
  where app 0 xs ys = ys
        app n xs E = xs
        app n (B x a b) (B y c d)
            | odd n     = B x (app m a (cons y d)) (app m b c)
            | otherwise = B x (app m a c) (app (m-1) b (cons y d))
          where m = half n

concat = foldr append empty
concatMap f = foldr (append . f) empty

take n xs = if n ≤ 0 then E else ta n xs
  where ta n E = E
        ta n (B x a b)
            | odd n     = B x (ta m a) (ta m b)
            | n == 0    = E
            | otherwise = B x (ta m a) (ta (m-1) b)
          where m = half n
```

Listing 2.11: Analysed functions of BraunSeq, part 1

```
-- not exported
combine E _ = E
combine (B x a b) c = B x c (combine a b)

drop n xs = if n ≤ 0 then xs else dr n xs
  where dr n E = E
        dr n t@(B x a b)
            | odd n     = combine (dr m a) (dr m b)
            | n == 0    = t
            | otherwise = combine (dr (m-1) b) (dr m a)
          where m = half n

splitAt i s = (take i s, drop i s)
subseq i len xs = take len (drop i xs)

reverse xs = rev00 (size xs) xs
  where
    rev00 n xs
      | n ≤ 1 = xs
    rev00 n (B x a b)
      | odd n     = let a'      = rev00 m a
                        (x',b') = rev11 m x b      in B x' a' b'
      | otherwise = let (x',a') = rev01 m a
                        b'      = rev10 (m-1) x b  in B x' b' a'
      where m = half n

    -- similarly rev11, rev01 and rev10

reverseOnto = append . reverse

zip (B x a b) (B y c d) = B (x,y) (zip a c) (zip b d)
zip _ _ = E

unzip E = (E, E)
unzip (B (x,y) a b) = (B x a1 b1, B y a2 b2)
  where (a1,a2) = unzip a
        (b1,b2) = unzip b
```

Listing 2.12: Analysed functions of BraunSeq, part 2

Chapter 3

New Implementations

3.1 QuickCheck Properties for Sequences

Edison already comes with a file `TestSeq.hs` that defines QuickCheck properties for all the sequence operations. Unfortunately, it has two flaws: It is monomorphic and it cannot check preservation of the structural invariant. That it is monomorphic means that all properties are defined on a type `Seq`, and what data structure this actually should be is defined by importing the respective module into `TestSeq.hs`, i.e. this file has to be changed whenever another implementation is to be tested. While checking a structural invariant could have been added to this by assuming the respective module also exports a property like `prop_inv :: Seq a → Bool` and then checking this property still holds after each modification, it seemed beneficial to redesign the whole thing to end up with a polymorphic function `checkseq` that performs all necessary tests for a data structure that can be defined via an argument.

To allow the invariant to be tested, we introduce a new type class that all sequences that are to be checked have to be instances of. Furthermore, we introduce a generator function that delivers an arbitrary sequence with given contents which will be used when checking the equality operator, see below.

```
class Sequence s ⇒ CheckableSequence s where
  invariant :: s a → Bool
  arbitraryWith :: [a] → Gen (s a)
```

It is very tempting now to write the ordinary `arbitrary` function in terms of `arbitraryWith`, i.e. as

```
arbitrary = arbitrary ≫= arbitraryWith
```

first generating an arbitrary list and then constructing a sequence containing its elements. There are two possible problems with this: First, when writing the `arbitrary` generators, care must be taken that they are complete, i.e. can construct every allowed instance of the data structure. Often, this is easier verified for an explicit `arbitrary` than for `arbitraryWith`, as the latter will typically be more complicated. Second, it may sometimes be beneficial to generate more elements than the size parameter suggests to avoid mostly trivial cases, especially in tree-like structures after `arbitrary` has already taken some recursive steps. If this should be done with the help of `arbitraryWith`, a lot of ahead-planning before the recursion would be needed, resulting in even more complication.

The properties themselves follow an approach that is somewhat different from what they looked like in `TestSeq.hs`. We first define a reference function in terms of `lview`, `empty` and `cons`, and possibly other functions that have been checked before, and then compare the results of the reference definition with that of the implementation to be tested and verify the invariant, in case a new sequence is among the results. It should be noted that `empty` and `cons` correspond to the two list constructors `[]` and `:` and furthermore, `lview` corresponds to the pattern matching possible on lists. This makes it fairly easy to adapt list representations of the functions as reference definitions, alas looking a little clumsier as the pattern matching has to be replaced with **case** statements.

Testing `lview`, `empty` and `cons` themselves looks a little different. For `lview`, we will assume it to be correct by definition, that is it gives the sequence interpretation to the underlying data structure by defining a head and a tail for it. `empty` and `cons` then have to be compatible with this interpretation which we will check without giving reference definitions for them.

However, we can check one thing about `lview`, namely that the tail it returns (if it does) fulfills the invariant:

```
prop_lview :: (CheckableSequence s, Eq a) ⇒ s a → Bool
prop_lview xs = case lview xs of
                    Nothing2 → True
                    Just2 _ xs' → invariant xs'
```

We can now verify that `empty` fulfills the invariant and that `lview empty` yields `Nothing2`. Unfortunately, we have to construct some way of telling the property the data type it should operate on, as we do not quantify over all sequences. We do this by explicitly passing in the invariant as an argument and use it to resolve the overloading. We do the same for the other constructor functions `single`, `formList`, `copy` and `tabulate`.

```
prop_empty :: (Sequence s, Eq a, Eq (s a)) ⇒ (s a → Bool) → Bool
prop_empty inv = inv xs && case lview xs of
                               Nothing2 → True
                               Just2 _ _ → False
  where xs = empty
```

There are two more things worthwhile noticing here. First, there is no variable at all we quantify over, which is perfectly ok with QuickCheck; it will just check the same thing 100 times. Second, the **where** clause is very important and will occur similarly throughout these properties. Replacing `xs` with `empty` in the **case** clause of the code fragment above would give a type error as the polymorphism could not be resolved to a specific type. By using the **where** clause we take advantage of the monomorphism restriction that forces `xs` to have the same type at every occurrence and let the type of the given invariant fix the type of the `xs`.

In order to check `cons`, equality testing is useful, and we can define it only in terms of `lview`, so we test it first. The reference definition performs element-wise equality testing by simultaneous recursion over both sequences.

```
equalsDef :: (Sequence s, Eq a) ⇒ s a → s a → Bool
equalsDef xs ys = case lview xs of
                    Nothing2 → case lview ys of
                                 Nothing2 → True
                                 Just2 _ _ → False
                    Just2 x xs' → case lview ys of
                                   Nothing2 → False
                                   Just2 y ys' → x==y && equalsDef xs' ys'
```

As no new sequence is generated, the resulting property is simply a comparison of both return values of reference definition and given implementation.

```
prop_equals :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → s a → Bool
prop_equals xs ys = (xs==ys) == equalsDef xs ys
```

In this way testing the equality operator poses yet another problem: While it would be sufficient to prove that

$$\forall \mathrm{xs}.\forall \mathrm{ys}.(\mathrm{xs} ==_{Impl} \mathrm{ys}) = (\mathrm{xs} ==_{Def} \mathrm{ys}),$$

we only perform automated testing, generating a limited number of instances for `xs` and `ys`. This makes it highly unlikely that two sequences are generated that are actually equal, except for very short cases. To compensate this, we will check `prop_equals` a second time with sequences generated such that they are equal. In order to be able to do so, we need a special generator that takes a list of elements and generates a sequence containing the respective elements, but with an arbitrary internal structure. This is why the `CheckableSequence` class contains the second function `arbitraryWith`.

Using equality, it is now easy to test that `cons` is correct by verifying that the resulting sequence satisfies the invariant and an `lview` undoes its effect.

```
prop_cons :: (CheckableSequence s, Eq a, Eq (s a)) ⇒ s a → a → Bool
prop_cons xs x = lview xxs == Just2 x xs && invariant xxs
  where xxs = cons x xs
```

Most of the remaining reference definitions and properties are straight-forward and do not require explanation. See appendix C for the complete source code.

As `lhead` may only be invoked for non-empty sequences, we have to ensure this by using QuickChecks conditional property feature. It would be nice if it was possible to check that `lhead` actually produces an error when invoked on an empty list, but this would unfortunately terminate the whole test-bench. (The same holds of course for the other functions requiring a non-empty sequence.)

```
prop_lhead :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → Property
prop_lhead xs = not (null xs) ⟹ lhead xs == lheadDef xs
```

To provide the user with some feedback whether his `arbitrary` function produces sequences of sufficient complexity, we `collect` the sizes of the test data when checking `size`.

```
prop_size  :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → Property
prop_size xs = collect (size xs) $ size xs == sizeDef xs
```

The `reduce` family of operations only return unambiguously defined results if the function argument represents an associative function, i.e. $x \oplus (y \oplus z) = (x \oplus y) \oplus z$. If this is the case, the result has to be equal to what a `fold` would yield, while the order in which the function is applied will differ. We can therefore simply use an appropriate `fold` as reference definition.

```
reducerDef :: Sequence s ⇒ (a → a → a) → a → s a → a
reducerDef = foldr
```

While QuickCheck provides a way to generate arbitrary functions which makes it possible to quantify over all functions for the fold operations, there is no (easy) way to quantify over all associative functions. To circumvent this, we will use only one such function, namely (++) and show that this is sufficient in the sense that if instead of testing the property, we would prove, we had then proven it for any associative function.

From what Wadler [Wad89] calls parametricity we can derive that for any two sets A, A' representing types, any function $a : A \to A'$ and any operator $\oplus'$ such that $a(x \oplus y) = (ax) \oplus' (ay)$, it holds that

$$a(\text{reducer } (\oplus)\ u\ xs) = \text{reducer } (\oplus')\ (au)\ (a^* xs)$$

where a^* is element-wise application of a (i.e. map a) (see appendix B). We now choose A to be $(A')^*$, i.e. lists of elements of A' and $(\oplus) = (++)$, the append operation. Furthermore, we define a as

$$a\ x = \begin{cases} (a\ x_1) \oplus' (a\ x_2) & \text{for } x = x_1 ++ x_2 \\ x' & \text{for } x = [x'] \end{cases}$$

With this definition, obviously $a(x \oplus y) = (ax) \oplus' (ay)$ if $\oplus'$ is associative so that we can leave the decomposition of multiple appends ambiguous without effecting the result. So we can use the above equivalence to derive results for arbitrary types and associative functions by just concentrating on appending and lists, starting with singletons, as obviously $a^{-1}\ x' = [x']$.

```
prop_reducer :: (Sequence s, Eq a, Eq (s a), Arbitrary a)
             ⇒ s a → a → Bool
prop_reducer xs' e' = let xs = map (λx' → [x']) xs'
                          e = [e']
                      in reducer (++) e xs == reducerDef (++) e xs
```

The same argument also holds for reducel and can easily be adapted for reduce1.

As we only use a generator for one type of element, we have to restrict the type of the arguments to the zip operations in the properties to be of the same type.

```
prop_zipWith :: (CheckableSequence s, Eq a, Eq (s a))
             ⇒ s a → s a → (a → a → a) → Bool
prop_zipWith xs ys f = invariant xys && xys == zipWithDef f xs ys
  where xys = zipWith f xs ys
```

As we will only be able to quantify explicitly over Seq a, we cannot generate the arguments to the unzip operations this way, but have to use implicit quantification. This however enforces us to resolve the type in another way, so we again take the invariant as an extra argument.

```
prop_unzip :: (Sequence s, Eq a, Eq (s a))
           ⇒ (s a → Bool) → s (a, a) → Bool
prop_unzip inv xys = inv xs && inv ys && (xs, ys) == unzipDef xys
  where (xs, ys) = unzip xys
```

To finally test all properties for a given data-type we have to collect them in a function that can be parameterized in the type it checks. Again, we just add the invariant as an additional argument just to resolve the overloading; to get meaningful results, the invariant given as an argument should always be the same as the one in the CheckableSequence class. The typical invocation will hence look like

```
checkseq (invariant :: Seq Int → Bool)
```

We start by checking that both `arbitrary` and `arbitraryWith` generate sequences that actually fulfill the invariant. This also defines the type of the generator `seqs`, that is enforced to be monomorphic throughout the function.

```
checkseq inv = do putStr "Checking␣arbitrary...␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs inv)
                  putStr "Checking␣arbitraryWith...␣␣␣␣␣"
                  quickCheck (λxs → forAllWith xs
                              (λs → property $ inv s && xs == toListDef s))
```

The remaining properties all follow one of the two following pattern, i.e. either explicit quantification or implicit quantification but with the invariant `inv` as an extra argument.

```
                  putStr "Checking␣lview␣...␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_lview)
                  putStr "Checking␣empty...␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (prop_empty inv)
                  putStr "Checking␣==␣...␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_equals)
```

With the exception of the equality test, where the special generator is used to generate sequences containing the same elements.

```
                  putStr "Checking␣==␣(equal␣cases)...␣␣"
                  quickCheck (λxs → forAllWith xs
                              (λs1 → forAllWith xs
                               (λs2 → property $ prop_equals s1 s2)))
```

When testing `concatMap`, we use a down-sized generator for the function argument to avoid producing too large sequences.

```
                  putStr "Checking␣concatMap...␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs
                              (λxs → forAll gen20
                               (λf → (prop_concatMap xs f))))
```

For the remining checks we refer to appendix C, but skip them here as they do not provide any new insights.

```
                where seqs = arbitrary
                      genWith = λxs → arbitraryWith xs
                      forAllWith = λxs prop → forAll (genWith xs) prop
                      gen20 :: Arbitrary a ⇒ Gen a
                      gen20 = sized (λn → resize (min 20 n) arbitrary)
```

All implementations described in the following contain a function

```
checkme = CheckSeq.checkseq (invariant :: Seq Int → Bool)
```

to simplify testing them, i.e. calling `checkme` is sufficient. As all sequence operations are polymorphic in the type of the element, it is easy to see by Wadler's parametricity theorem that if a property for them holds for some element type `a`, then it holds for any type of same or lower cardinality. As we furthermore only perform a limited number of tests, the difference between types with infinite domains and those with sufficiently large, but finite domains (like `Int`) becomes irrelevant. This justifies why we only test for an element type of `Int`, but are still confident that the properties hold for other types as well.

3.2 SimpleDeque

The simple deque implementation suggested by Hoogerwoord [Hoo92b] is very similar to the `SimpleQueue`, and therefore has very similar properties, i.e. constant amortised time bounds for the typical deque operations if used non-persistently. The data type also consists of two lists, one front and one rear list.

```
data Seq a = D [a] [a]
```

Contrary to the `SimpleQueue`, the two lists are handled symmetrically, and the maintained invariant is that whenever the deque contains at least two elements, both lists are non-empty. However, to reduce the number of cases that must be distinguished, we arbitrarily demand that the element of a singleton deque is stored in the front list.

```
invariant :: Seq a → Bool
invariant (D [] []) = True          -- empty
invariant (D [_] []) = True         -- one element
invariant (D (_:_) (_:_)) = True    -- two or more elements
invariant (D _ _) = False
```

The runtime analysis is carried out assuming strictness and applying the physicist's method for amortisation, where the potential is defined to be the difference of the length of the two lists. Hoogerwoord showed that with this potential, `empty`, `ltail`, `rtail`, `cons`, `snoc` and `reverse` all run in $\mathcal{O}(1)$ amortised.

Except for `ltail` and `rtail`, those operations are almost literally translated into Haskell from Hoogerwoords article. For the two `tails`, we first define a function that, given a front and a rear list, creates a deque that fulfils the invariant.

```
makeD xs [] = D xs' (L.reverse ys')
  where (xs', ys') = L.splitAt ((L.size xs + 1) `div` 2) xs
makeD [] ys = D (L.reverse xs') ys'
  where (ys', xs') = L.splitAt (L.size ys `div` 2) ys
makeD xs ys = D xs ys
```

The first two cases handle the situation of (at least) one list being empty, where the other list is split into half and one of the halves is reversed. If the length of the list is odd, the split is performed such that the front list gets one element longer than the rear to handle the singleton case as demanded. The runtime complexity of this obviously is linear in the length of the non-empty list, the resulting potential is at most one. In the third case both lists are non-empty and thus can be used unaltered, what requires constant time. At first, the calculation of the size of the non-empty list seems to be avoidable by maintaining separate size fields in the data-type. This approach would, however, introduce an additional overhead to all operations, thereby effectively decreasing the overall performance if mainly operations are used that add or remove single elements.

The two basic constructors are straight-forward and obviously run in $\mathcal{O}(1)$ and result in a potential of zero or one, respectively.

```
empty = D [] []
single x = D [x] []
```

The `cons` and `snoc` operations are similarly easy to implement.

```
cons x (D xs []) = D [x] xs         -- xs contains zero or one element
cons x (D xs ys) = D (x:xs) ys

snoc (D [] _) y = D [y] []          -- ys is empty
snoc (D xs ys) y = D xs (y:ys)
```

If both lists are non-empty, the new element is added to the respective list, taking constant time and increasing the potential by at most 1. If the rear list is empty when `cons` is invoked, the old front list becomes the new rear list (reversing is unnecessary as the front list has at most one element) and the new front list is the singleton list containing the new element. For an empty front list when `snoc` is invoked, things are even easier as the rear list is guaranteed to be empty. Hence, these cases also require constant time and increase the potential by at most one.

The `append` operation distinguishes three cases, depending on the second deque.

```
append d1 (D [] _) = d1
append d1 (D [x2] []) = snoc d1 x2
append (D xs1 ys1) (D xs2 ys2) =
    D (xs1 ++ L.reverseOnto ys1 xs2) ys2
```

If the second deque contains no or only one element, the `append` degenerates to identity or `snoc`, respectively, yielding constant runtime. The non-trivial case obviously requires $\mathcal{O}(n_1)$ steps in total, $|\text{xs1}|$ for the append and $|\text{ys1}|$ for the `reverseOnto`, with $n_1 = |\text{xs1}| + |\text{ys1}|$ being the size of the first deque. For the initial potential we have $\Phi(n) = ||\text{xs1}| - |\text{ys1}|| + ||\text{xs2}| - |\text{ys2}||$, while the resulting potential is

$$\Phi(n+1) = |(|\text{xs1}| + |\text{ys1}| + |\text{xs2}|) - |\text{ys2}|| \leq ||\text{xs1}| + |\text{ys1}|| + ||\text{xs2}| - |\text{ys2}|| = n_1 + ||\text{xs2}| - |\text{ys2}||\,,$$

so the change in potential is bounded by

$$\Phi(n+1) - \Phi(n) \leq n_1 - ||\text{xs1}| - |\text{ys1}|| \leq n_1,$$

giving an amortised time complexity of $\mathcal{O}(n_1)$.

Accessing the left-most element requires distinction between three cases.

```
lview (D [] _) = Nothing2
lview (D [x] ys) = Just2 x (makeD [] ys)
lview (D (x:xs) ys) = Just2 x (D xs ys)

lhead (D [] _) = error "SimpleDeque.lhead:␣empty␣sequence"
lhead (D (x:_) _) = x

ltail d@(D [] _) = d
ltail (D [x] ys) = makeD [] ys
ltail (D (x:xs) ys) = D xs ys
```

If the front list is empty, the whole deque is empty. If the front list contains exactly one element, this is the demanded one, but calculating the tail requires a call to `makeD` to reestablish the invariant. (As `lhead` does not produce a tail, it does not distinguish this case.) Otherwise, the front list contains at least two elements, hence taking the tail of the list as new front will not contradict the invariant. Except for the second case, the operations have amortised runtime of $\mathcal{O}(1)$, as they obviously require constant runtime and change the potential by at most one. The second case needs special attention. The call to `makeD` has runtime complexity $\mathcal{O}(n)$, but the potential is reduced from $|1-(n-1)| = n-2$ to 0 or 1, i.e. at least by $n-3$, compensating the linear runtime, thereby giving an amortised runtime $\mathcal{O}(1)$.

Accessing the right-most element is very similar.

```
rview (D [] _) = Nothing2
rview (D [x] []) = Just2 (D [] []) x
rview (D xs (y:ys)) = Just2 (makeD xs ys) y

rhead (D [] _) = error "SimpleDeque.rhead:␣empty␣sequence"
rhead (D [x] []) = x
rhead (D _ (y:_)) = y

rtail (D _ []) = D [] []   -- if rear is empty, at most one element in deque
rtail (D xs (_:ys)) = makeD xs ys
```

Although the distinguished cases are slightly different, due to the asymmetrical handling of singleton deques, the above argument can easily be adapted to establish constant amortised runtime for these operations, too.

Checking whether a deque contains any elements at all obviously run in constant time.

```
null (D [] _) = True
null _ = False
```

Calculation of the size of the deque requires determining the length of both lists, resulting in a runtime complexity of $\mathcal{O}(n)$.

```
size (D xs ys) = length xs + length ys
```

Thanks to the symmetry, reversing a deque is fairly simple.

```
reverse d@(D xs []) = d
reverse (D xs ys) = D ys xs
```

If the deque contains at least two elements, both list are non-empty and can thus simply be exchanged. Otherwise, the deque contains zero or one element and is therefore equal to its reverse. Both cases obviously run in constant time and leave the potential unaltered, yielding constant amortised runtime.

The `reverseOnto` operation is very similar to `append`.

```
reverseOnto d1@(D xs1 ys1) (D [] _) = reverse d1
reverseOnto d1@(D xs1 ys1) (D [x2] []) = snoc (reverse d1) x2
reverseOnto (D xs1 ys1) (D xs2 ys2) =
    D (ys1 ++ L.reverseOnto xs1 xs2) ys2
```

Considering that `reverse` runs in $\mathcal{O}(1)$ and that the non-trivial case is equal to `append` except for the exchange of `xs1` and `ys1`, it is easily seen that with an argument very similar to that for `append`, the amortised runtime turns out to be $\mathcal{O}(n_1)$.

Using the helper function, conversion of a list to a deque is straight forward, as is the conversion back to a list.

```
fromList xs = makeD xs []

toList (D xs []) = xs
toList (D xs ys) = xs ++ L.reverse ys
```

Both operations clearly take $\mathcal{O}(n)$ time, but is should be noted that `toList` returns a result that can be calculated incrementally up to the end of the front list in a lazy manner.

The `map` and `fold`/`reduce`-family of operations are simply performed by calling the appropriate list functions on the front and rear list. The required runtime is mainly determined by applying the function argument `f` n times.

```
map f (D xs ys) = D (L.map f xs) (L.map f ys)

foldr f e (D xs ys) = L.foldr f (L.foldl (flip f) e ys) xs

foldl f e (D xs ys) = L.foldr (flip f) (L.foldl f e xs) ys

foldr1 f (D [] _) = error "SimpleDeque.foldr1:␣empty␣sequence"
foldr1 f (D [x] []) = x
foldr1 f (D xs ys) = L.foldr f (L.foldl1 (flip f) ys) xs

foldl1 f (D [] _) = error "SimpleDeque.foldl1:␣empty␣sequence"
foldl1 f (D xs ys) = L.foldr (flip f) (L.foldl1 f xs) ys

reducer f e (D xs ys) = L.reducer f (L.reducel (flip f) e ys) xs

reducel f e (D xs ys) = L.reducer (flip f) (L.reducel f e xs) ys

reduce1 f (D [] _) = error "SimpleDeque.reduce1:␣empty␣sequence"
reduce1 f (D xs ys) = L.reducer (flip f) (L.reduce1 f xs) ys
```

Splitting a deque at a specified position can be performed by repeated application of `lview` and `cons`. As both `lview` and `cons` have constant amortised runtime and are called i times, the total amortised runtime is $\mathcal{O}(i)$.

```
splitAt _ d@(D [] []) = (d, d)
splitAt i d | i≤0      = (empty, d)
            | otherwise = let Just2 x d' = lview d
                              (d1, d2) = splitAt (i-1) d'
                          in (x `cons` d1, d2)
```

The `copy` constructor is easily implemented using the `copy` for lists to generate front and end lists with suitable sizes.

```
copy n x = D (L.copy ((n+1) `div` 2) x) (L.copy (n `div` 2) x)
```

The required runtime is $\mathcal{O}(n)$ and the potential of the resulting deque is at most 1, do the amortised runtime is $\mathcal{O}(n)$.

Similarly, `tabulate` can be implemented by constructing a deque with appropriately enumerated lists and then applying `map` to it.

```
tabulate n f | n ≤ 0     = empty
             | otherwise = let k=(n+1) ‘div‘ 2 - 1
                           in  map f (D [0..k] [n-1, n-2..k+1])
```

Application of `f` n times is again the predominant part of the required runtime.

Like `map`, the `filter` and `partition` operations can be carried out directly by using the respective list operations. But as this my cause one of the involved lists to become empty, a call to `makeD` is required. Both the runtime of the `makeD` and the possible increase of potential are$\mathcal{O}(n)$, as is the number of applications of `p`.

```
filter p (D xs ys) = makeD (L.filter p xs) (L.filter p ys)

partition p (D xs ys)
  = (makeD xsT ysT, makeD xsF ysF)
 where
   (xsT,xsF) = L.partition p xs
   (ysT,ysF) = L.partition p ys
```

The check whether a certain index is within the deque can be done by checking whether it is within the catenation of the front and rear list; reversing the rear is not necessary as it does not change its length. Observing the incrementality of the `++`, this implementation has time complexity $\mathcal{O}(i)$.

```
inBounds (D xs ys) = L.inBounds (xs ++ ys)
```

The remaining operations are all implemented using defaults.

Concatenating multiple deques is performed by right-associatively `append`ing them using `foldr`.

```
concat = concatUsingFoldr
concatMap = concatMapUsingFoldr
```

The `foldr` runs in $\mathcal{O}(n)$ plus the time needed for the `append`s, which is $\mathcal{O}(m)$ with m being the length of the resulting deque, plus the time needed for the `map` for `concatMap`, giving a total runtime of $\mathcal{O}(n + m)$.

Like `splitAt`, also `take` and `drop` can be implemented by repeated calls of `lview` or `ltail`, respectively. As `lview` and `ltail` run in constant amortised time, this obviously results in an amortised time complexity of $\mathcal{O}(i)$.

```
take = takeUsingLview
drop = dropUsingLtail
```

Using `drop`, it is easy to implement `lookup` and derivatives.

```
lookupM = lookupMUsingDrop
lookup = lookupUsingLookupM
lookupWithDefault = lookupWithDefaultUsingLookupM
```

Unfortunately, as the original deque may be still be used afterwards, the amortised runtime analysis that has been done for `ltail` and hence for `drop` and relied on non-persistent usage cannot be transferred to these operations. Instead, if the index happens to lie in the rear list, the operations always take $\mathcal{O}(n)$.

This is not the case for `adjust` and `update`, which are implemented by splitting the deque, updating the element and appending the two parts again.

```
adjust = adjustUsingSplitAt
update = updateUsingAdjust
```

As both the splitting and the appending take time $\mathcal{O}(i)$ amortised, this also is the total amortised time complexity.

The *`WithIndex`-operations utilise the `toList` function. As the conversion to a list (and back in case of `map`) is $\mathcal{O}(n)$, applying `f` n times determines the runtime of these operations.

```
mapWithIndex = mapWithIndexUsingLists
foldrWithIndex = foldrWithIndexUsingLists
foldlWithIndex = foldlWithIndexUsingLists
```

Using `drop` and `take`, it is easy to implement `subseq`. The total amortised time complexity is that of `drop` plus that of `take`, i.e. $\mathcal{O}(i + j)$ (for `subseq i j`).

```
subseq = subseqDefault
```

The *`While` operations traverse the deque by repeated application of `lview`. Per accepted element, this takes constant amortised time for `lview`, and `cons` if applicable, plus the time required for evaluation of the predicate, where at most n elements can be accepted.

```
takeWhile = takeWhileUsingLview
dropWhile = dropWhileUsingLview
splitWhile = splitWhileUsingLview
```

The `zip`/`unzip`-operations all operate by conversion to lists internally.

```
zip = zipUsingLists
zip3 = zip3UsingLists
zipWith = zipWithUsingLists
zipWith3 = zipWith3UsingLists
unzip = unzipUsingLists
unzip3 = unzip3UsingLists
unzipWith = unzipWithUsingLists
unzipWith3 = unzipWith3UsingLists
```

For the `zip`s, this means that the longest deque determines the runtime if its front list is shorter than the shortest deque so that the `reverse` is forced. Assume that the length of the longer deque n_2 is more than three times that of the shorter deque n_1, as otherwise, $n_2 \leq 3n_1 = \mathcal{O}(n_1)$. But then, for the front list to be shorter than the shorter deque, the potential of the longer deque has to be at least $(n_2 - n_1) - n_1 = n_2 - 2n_1$, where $n_2 - n_1$ is the minimum length of the rear list and n_1 is the maximum length of the front list. As the potential of the resulting deque is at most 1, it is decreased by at least $n_2 - 2n_1 = \mathcal{O}(n_2)$, so the amortised runtime is $\mathcal{O}(\min(n_1, n_2))$. The same argument holds for the case of three deques. The amortised runtime complexity of the `unzip`s is trivially $\mathcal{O}(n)$.

Generating an arbitrary `SimpleDeque` can be performed by generating arbitrary front and rear lists. Only if one of them is empty, special attention is required not to violate the invariant.

Specifically, the empty list is made the rear list and the other list is shortened to at most one element. I.e. if both lists are empty, the resulting deque is empty; if only one list is empty, the resulting deque contains exactly one element; if both lists are non-empty (but otherwise of arbitrary length), all their elements are preserved. This way, every possible SimpleDeque can be generated: the empty case if both generated lists are empty, the one-element case if exactly one list is empty, or an arbitrary distribution of elements among front and rear list if both lists are non-empty and the resulting queue therefore contains at least two elements.

```
instance Arbitrary a ⇒ Arbitrary (Seq a) where
  arbitrary = do xs ← arbitrary
                 ys ← arbitrary
                 return (if L.null xs then D (L.take 1 ys) []
                         else if L.null ys then D (L.take 1 xs) []
                         else D xs ys)
```

Generating a SimpleDeque for a given list of elements distinguishes two cases: If less than two elements (i.e. zero or one) are passed in, they are all stored in the front list, which is directly enforced by the invariant. Otherwise, the input list at split at a random position such that both parts are non-empty to acquire the front and rear list, respectively. Again it is easy to see that any possible SimpleDeque can be generated, as for the non-trivial case with more than one element, the splitting is done at an arbitrary position in the allowed range.

```
instance CheckSeq.CheckableSequence Seq where
  arbitraryWith xs = let len = S.size xs
                     in if len < 2 then return (D xs [])
                        else do lenf ← choose (1, len-1)
                                let (f, r') = L.splitAt lenf xs
                                return (D f (L.reverse r'))
```

3.3 BankersDeque

Similarly to how SimpleDeque could be based on SimpleQueue by adding symmetry, we can get a deque implementation suitable for persistent usage by adapting the BankersQueue to maintain a symmetric invariant [Oka95c, Oka98]. The data type of the deque is equal to that of the queue, with a front and a rear list and their respective lengths.

```
data Seq a = D !Int [a] [a] !Int
```

While for the BankersQueue the invariant was that the front list had to be at least as long as the rear, for the BankersDeque, the length have to be balanced in some sense; in particular, the maintained invariant is

$$|\mathrm{f}| \leq c|\mathrm{r}| + 1 \wedge |\mathrm{r}| \leq c|\mathrm{f}| + 1$$

for some integer constant $c > 1$. Note that this implies that for any deque containing at least two elements, both lists are non-empty. Specified in Haskell itself, the invariant looks as follows:

```
invariant :: Seq a → Bool
invariant (D lenf f r lenr) = lenf == L.size f
                              && lenr == L.size r
                              && lenf ≤ c*lenr+1
                              && lenr ≤ c*lenf+1
```

Many operations will have amortised runtime that increases with c, so we choose

```
c = 2::Int
```

Higher values of c have the advantage that for sequences of accesses to same end of the deque, fewer rotations will be executed. However, in these situations, it may be assumed that one of the queue implementations may be more suitable anyway, so we use the lowest possible value for c.

The amortised runtime analysis, with which Okasaki shows contant runtime for `cons`, `snoc`, `ltail`, `rtail` and `reverse`, uses the bankers method where for both lists, $d(i)$ denotes the number of debits on the ith element and $D(i) = \sum_{j=0}^{i}$ is the total debit for the first $i+1$ elements. The maintained debit invariant is

$$D(i) \leq \min\left(i(c+1), cs+1-t\right), \quad s = \min(|f|, |r|), \quad t = \max(|f|, |r|).$$

Note that this enforces $D(i) = 0$, thus also $d(i) = 0$, i.e. the list heads are free of debits and can be accessed at will.

We start by defining a helper function (slightly adapted from the `check` function in [Oka98]) that will (re)establish the structural invariant by truncating the longer list to half the combined length of both list and reversing the rest onto the back of the shorter list.

```
makeD lenf f r lenr
  | lenf>c*lenr+1 = let i = (lenf+lenr) `div` 2
                        j = lenf+lenr-i
                    in D i (L.take i f) (r ++ L.reverse (L.drop i f)) j
  | lenr>c*lenf+1 = let j = (lenf+lenr) `div` 2
                        i = lenf+lenr-j
                    in D i (f ++ L.reverse (L.drop j r)) (L.take j r) j
  | otherwise     = D lenf f r lenr
```

The two basic constructors both trivially maintain both the structural and the debit invariant and run in constant time.

```
empty = D 0 [] [] 0
single x = D 1 [x] [] 0
```

Using the helper function, `cons` and `snoc` are equally easy to implement, as found in [Oka98].

```
cons x (D lenf f r lenr) = makeD (lenf+1) (x:f) r lenr
snoc (D lenf f r lenr) y = makeD lenf f (y:r) (lenr+1)
```

For the runtime analysis of `cons`, we have to distinguish whether the call of `makeD` causes a rotation or not. If not, $|\mathrm{f}|$ is increased by one, thereby violating the invariant for all nodes where previously $D(i) = c\min(|\mathrm{f}|, |\mathrm{r}|) + 1 - \max(|\mathrm{f}|, |\mathrm{r}|)$, if $|\mathrm{f}| > |\mathrm{r}|$. To reestablish the invariant, it is obviously sufficient to discharge one debit per list, say the first one. In case a rotation is forced, we know that before the `cons`, $|\mathrm{f}| = c|\mathrm{r}|+1$ and so $c\min(|\mathrm{f}|, |\mathrm{r}|)+1-\max(|\mathrm{f}|, |\mathrm{r}|) = c|\mathrm{r}|+1-|\mathrm{f}| = 0$, i.e. all debits had been discharged. After the rotation, there is one debit on every element in the front list and one debit on the first $|\mathrm{r}|$[1] elements on the rear list (as `take` and `append` for lists are incremental), plus $|\mathrm{f}|+1 = c|\mathrm{r}|+2$ on the $|\mathrm{r}|$th node of the rear list. So the resulting debits are

$$D_{\mathrm{f}'}(i) = i+1 \qquad D_{\mathrm{r}'}(i) = \begin{cases} i+1 & \text{for } i < |\mathrm{r}| \\ (c+1)|\mathrm{r}|+2 & \text{for } i \geq |\mathrm{r}|. \end{cases}$$

[1]`f` and `r` still refer to the lists before the `cons` while `f'` and `r'` denote the lists after the rotation.

The debit invariant can therefore be reestablished by discharging the one debit on the head elements of both lists and discharging one extra debit on the rear list, giving constant amortised runtime.

By symmetry, the same holds for `snoc`.

Thanks to the explicitly stored sizes, we can perform `append` in time linear to the shorter of the two deques.

```
append (D lenf1 f1 r1 lenr1) (D lenf2 f2 r2 lenr2)
    | lenf1+lenr1 ≤ lenf2+lenr2 =
          makeD (lenf1 + lenr1 + lenf2) (f1 ++ L.reverseOnto r1 f2) r2 lenr2
    | otherwise =
          makeD lenf1 f1 (r2 ++ L.reverseOnto f2 r1) (lenr2 + lenf2 + lenr1)
```

For the analysis, we will assume that $|\mathrm{f1}| + |\mathrm{r1}| = n_1 \leq n_2 = |\mathrm{f2}| + |\mathrm{r2}|$ and establish that it has amortised runtime complexity $\mathcal{O}(n_1)$, then, by symmetry, `append` will obviously run in $\mathcal{O}(\min(n_1, n_2))$ amortised.

First, we discharge all debits on `f1` and `r1`, which are bounded by $\mathcal{O}(n_1)$, and furthermore pay the costs for the `++` and `reverseOnto`, which are together also $\mathcal{O}(n_1)$. What remains to do is similar to the analysis of `cons`, again we distinguish whether a rotation is performed or not. If not, like with `cons`, it is sufficient to discharge n_1 debits on both resulting lists. If a rotation is caused, we know that $|\mathrm{f2}| + x = c|\mathrm{r2}| + 1$ for some $x < n_1$, so the debits on `f2` and `r2` are bounded by

$$\begin{aligned} c\min(|\mathrm{f2}|, |\mathrm{r2}|) + 1 - \max(|\mathrm{f2}|, |\mathrm{r2}|) &\leq c|\mathrm{r2}| + 1 - |\mathrm{f2}| \\ &= c|\mathrm{r2}| + 1 - (c|\mathrm{r2}| + 1 - x) \\ &= x < n_1, \end{aligned}$$

and so can be discharged in $\mathcal{O}(n_1)$. With an argument similar to the one for `cons`, we can establish that the debits after the rotation are

$$D_{\mathrm{f}'}(i) = i + 1 \qquad D_{\mathrm{r}'}(i) = \begin{cases} i + 1 & \text{for } i < |\mathrm{r}| \\ (c+1)|\mathrm{r}| - x + 1 + n_1 & \text{for } i \geq |\mathrm{r}|. \end{cases}$$

So by discharging one debit on the front list and $n_1 + 1 - x = \mathcal{O}(n_1)$ debits on the rear list the debit invariant can be reestablished, giving a total amortised runtime complexity of $\mathcal{O}(n_1)$.

Accessing the first (last) element is straight-forward and requires distinction of three cases: the deque may be empty, may contain one element in the rear list (front list) or has a non-empty front list (rear list). Note: The Haskell implementations given in [Oka98] are incorrect as they are missing the second case; the Standard ML ones are correct, though.

```
lview (D _ [] [] _) = Nothing2
lview (D _ [] [y] _) = Just2 y empty
lview (D lenf (x:f) r lenr) = Just2 x (makeD (lenf-1) f r lenr)

lhead (D _ [] [] _) = error "BankersDeque.lhead:␣empty␣sequence"
lhead (D _ [] [y] _) = y
lhead (D _ (x:_) _ _) = x

ltail (D lenf (_:f) r lenr) = makeD (lenf-1) f r lenr
ltail _ = empty                    -- zero or one element
```

```
rview (D _ [] [] _) = Nothing2
rview (D _ [x] [] _) = Just2 empty x
rview (D lenf f (y:r) lenr) = Just2 (makeD lenf f r (lenr-1)) y

rhead (D _ [] [] _) = error "BankersDeque.rhead:␣empty␣sequence"
rhead (D _ [x] [] _) = x
rhead (D _ _ (y:_) _) = y

rtail (D lenf f (_:r) lenr) = makeD lenf f r (lenr-1)
rtail _ = empty                    -- zero or one element
```

Thanks to the symmetry, we can restrict our analysis again to the `lview`-family and the same results will also hold for the `rview`-family. As the head element of the lists are guaranteed to be free of debits, `lhead` clearly runs in constant time, as do `lview`/`ltail` in case the returned deque is empty. If the returned deque is non-empty, we have to distinguish again whether a rotation is required or not. If not, the debit invariant in the front list may be violated due to the resulting index shift. This can be countered by discharging the first $c+1$ debits in the list. Furthermore, $\min(|\mathrm{f}|, |\mathrm{r}|)$ may have been reduced by one, making it necessary to discharge c debits on the rear list. In case a rotation is required, before the `lview`/`ltail` it must have been the case that $c|\mathrm{f}|+1 = |\mathrm{r}|$ and all debits had been discharged. After the rotation, there is one debit on every element of the rear list, one debit on the first $|\mathrm{f}|-1$ elements of the front list and $|\mathrm{r}| = c|\mathrm{f}|+1$ debits on the $(|\mathrm{f}|-1)$th element, giving the debits

$$D_{\mathrm{r}'}(i) = i+1 \qquad D_{\mathrm{f}'}(i) = \begin{cases} i+1 & \text{for } i < |\mathrm{f}|-1 \\ (c+1)|\mathrm{f}|+1 & \text{for } i \geq |\mathrm{f}|-1. \end{cases}$$

The debit invariant can be reestablished by discharging the debit on the head of the rear list and discharging the first $c+2$ debits on the front list. Thus, the amortised runtime complexity is $\mathcal{O}(1)$.

Determining the size of a deque is simple due to the explicit size fields and obviously runs in $\mathcal{O}(1)$.

```
null (D lenf _ _ lenr) = (lenf == 0) && (lenr == 0)
size (D lenf _ _ lenr) = lenf + lenr
```

Reversing a deque is equally simple, as both the structural and the debit invariant are symmetric and are thus preserved when front and rear list are exchanged, making this operation run in constant time.

```
reverse (D lenf f r lenr) = D lenr r f lenf
```

Conversion from a list can be performed by a simple call to the helper function after first determining the length of the list.

```
fromList xs = makeD (length xs) xs [] 0
```

Determining the length takes $|\mathrm{xs}|$ steps, and after the rotation that will be necessary for $|\mathrm{xs}| > 1$, there are one debit on the first element in the front list and $|\mathrm{xs}|$ debits on the head of the rear list, which have to be discharged, resulting in an amortised runtime of $\mathcal{O}(|\mathrm{xs}|)$.

Conversion to a list has to discharge all $\mathcal{O}(n)$ debits on the deque and then append the reversed rear to the front list, requiring additional n steps, giving a total amortised runtime complexity of $\mathcal{O}(n)$.

```
toList (D _ f [] _) = f
toList (D _ f r  _) = f ++ L.reverse r
```

Like for the `SimpleDeque`, the `map` and `fold`/`reduce`-family of operations are simply performed by calling the appropriate list functions on the front and rear list. The required runtime is mainly determined by applying the function argument `f` n times.

```
map f (D i xs ys j) = D i (L.map f xs) (L.map f ys) j

foldr f e (D _ xs ys _) = L.foldr f (L.foldl (flip f) e ys) xs

foldl f e (D _ xs ys _) = L.foldr (flip f) (L.foldl f e xs) ys

foldr1 _ (D _ []  [] _) = error "BankersDeque.foldr1:␣empty␣sequence"
foldr1 _ (D _ [x] [] _) = x
foldr1 f (D _ xs  ys _) = L.foldr f (L.foldl1 (flip f) ys) xs

foldl1 _ (D _ [] []  _) = error "BankersDeque.foldl1:␣empty␣sequence"
foldl1 _ (D _ [] [y] _) = y
foldl1 f (D _ xs ys  _) = L.foldr (flip f) (L.foldl1 f xs) ys

reducer f e (D _ xs ys _) = L.reducer f (L.reducel (flip f) e ys) xs

reducel f e (D _ xs ys _) = L.reducer (flip f) (L.reducel f e xs) ys

reduce1 _ (D _ [] []  _) = error "BankersDeque.reduce1:␣empty␣sequence"
reduce1 _ (D _ [] [y] _) = y
reduce1 f (D _ xs ys  _) = L.reducer (flip f) (L.reduce1 f xs) ys

mapWithIndex f (D i xs ys j) =
    D i (L.mapWithIndex f xs) (L.mapWithIndex f' ys) j
    where f' idx x = f (i+j-idx-1) x

foldrWithIndex f e (D i xs ys j) =
    L.foldrWithIndex f (L.foldlWithIndex f' e ys) xs
    where f' y idx x = f (i+j-idx-1) x y

foldlWithIndex f e (D i xs ys j) =
    L.foldrWithIndex f' (L.foldlWithIndex f e xs) ys
    where f' idx x y = f y (i+j-idx-1) x
```

Similarly, the `copy` and `tabulate` constructors can be implemented by constructing the two lists appropriately.

```
copy n x
  | n < 0     = empty
  | otherwise = let lenf = n `div` 2
                    lenr = n - lenf
                in D lenf (L.copy lenf x) (L.copy lenr x) lenr
tabulate n f
  | n < 0     = empty
  | otherwise = let lenf = n `div` 2
                    lenr = n - lenf
                in D lenf (L.map f [0..lenf-1]) (L.map f [n-1,n-2..lenf]) lenr
```

As for both operations the lists can be build incrementally, one debit will be placed on every element (assuming for `tabulate` that `f` runs in constant time), so that the debit invariant can be established by discharging only one debit per list (on the head element), curiously yielding constant amortised runtime.

Like `map`, the `filter` and `partition` operations can be carried out directly by using the respective list operations. But as this my cause the structural invariant to be violated, a call to `makeD` is required. The amortised runtime is obviously $\mathcal{O}(n)$, assuming that `p` runs in constant time.

```
filter p (D lenf f r lenr) =
    let f' = L.filter p f
        r' = L.filter p r
    in makeD (L.size f') f' r' (L.size r')

partition p (D lenf f r lenr)
  = (makeD (L.size fT) fT rT (L.size rT), makeD (L.size fF) fF rF (L.size rF))
 where
   (fT,fF) = L.partition p f
   (rT,rF) = L.partition p r
```

Looking up an element at a certain index is carried out by directly looking up the element in the respective list, which can be determined using the explicit size fields.

```
lookup (D lenf f r lenr) i
  | i < lenf  = L.lookup f i
  | otherwise = L.lookup r (lenr - (i - lenf) - 1)

lookupM (D lenf f r lenr) i
  | i < lenf  = L.lookupM f i
  | otherwise = L.lookupM r (lenr - (i - lenf) - 1)

lookupWithDefault d (D lenf f r lenr) i
  | i < lenf  = L.lookupWithDefault d f i
  | otherwise = L.lookupWithDefault d r (lenr - (i - lenf) - 1)
```

If the index is in the front list, the debits on the first i elements have to be discharged, which are bounded by $\mathcal{O}(i)$, and the `lookup` in the list also takes $\mathcal{O}(i)$ time. If the index lies in the rear list, the worst case occurs for $i = |\mathrm{f}|$, which results in looking up the last element in the rear list, which makes it necessary to discharge all debits plus requires $|\mathrm{r}|$ time. But as $|\mathrm{r}| \leq c|\mathrm{f}| + 1 = ci + 1 = \mathcal{O}(i)$, we can establish amortised time complexity $\mathcal{O}(i)$ for both cases.

The same holds for `update` and `adjust` (plus the time to execute `g` for `adjust`). Actually, the cost for the list-`update`/`adjust` could be assigned to new debits, but this would not change the asymptotic time complexity, as the old $\mathcal{O}(i)$ debits would have to be discharged before, anyway.

```
update i e d@(D lenf f r lenr)
  | i < lenf  = if i < 0 then d
                else D lenf (L.update i e f) r lenr
  | otherwise = let k' = lenr - (i - lenf) - 1
```

```
                 in if k' < 0 then d
                    else D lenf f (L.update k' e r) lenr
adjust g i d@(D lenf f r lenr)
  | i < lenf  = if i < 0 then d
                 else D lenf (L.adjust g i f) r lenr
  | otherwise = let k' = lenr - (i - lenf) - 1
                 in if k' < 0 then d
                    else D lenf f (L.adjust g k' r) lenr
```

The take operation also distinguishes whether the cut-off index is in the front or the rear list and then calls the appropriate list function.

```
take len d@(D lenf f r lenr)
  | len ≤ 0 = empty
  | len ≤ lenf = makeD len (L.take len f) [] 0
  | len ≤ lenf+lenr = let len' = len - lenf
                       in makeD lenf f (L.drop (lenr - len') r) len'
  | otherwise = d
```

We analyse the two cases separately. If $\text{len} \leq |\text{f}|$, both the list-take and the caused rotation obviously run in $\mathcal{O}(\text{len})$. Otherwise, as argued for the lookup-functions, the list-drop also runs in $\mathcal{O}(\text{len})$, leaving the cost for makeD to be analysed. If no rotation is forced, all debits on the rear list have already been discharged for the drop and its cost has been paid, leaving the front list, from which $c(\text{len} - |\text{f}|) = \mathcal{O}(\text{len})$ must be discharged to maintain the debit invariant, as $\min(\text{f}, \text{r})$ may be reduced by up to $(\text{len} - |\text{f}|)$. If a rotation is performed, we know that for the original deque it is the case that $c(|\text{r}| - x) + 1 = |\text{f}|$ for some $x < \text{len} - |\text{f}|$ (not necessarily integer). Thus, the debits present on the front list before the rotation are bounded by $c|\text{r}| + 1 - |\text{f}| = c|\text{r}| + 1 - (c(|\text{r}| - x) + 1) = cx < c(\text{len} - |\text{f}|) = \mathcal{O}(\text{len})$ and can be discharged. After the rotation, the debits on the two lists are

$$D_{\text{f}'}(i) = i + 1 \qquad D_{\text{r}'}(i) = \begin{cases} i+1 & \text{for } i < \text{len} - |\text{f}| \\ \text{len} & \text{for } i \geq \text{len} - |\text{f}|. \end{cases}$$

So, furthermore discharging the two debits on the list heads and additional len debits on the rear list reestablish the debit invariant, yielding $\mathcal{O}(\text{len})$ amortised runtime complexity.

By symmetry, we can then easily establish $\mathcal{O}(n - \text{len})$ runtime for drop.

```
drop len d@(D lenf f r lenr)
  | len ≤ 0 = d
  | len ≤ lenf = makeD (lenf - len) (L.drop len f) r lenr
  | len ≤ lenf+lenr = let len' = len - lenf
                       in  makeD 0 [] (L.take (lenr - len') r) (lenr - len')
  | otherwise = empty
```

We want to establish runtime $\mathcal{O}(\text{len})$, however. For the case of the index len being in the rear list, this is easy, as again $\mathcal{O}(n) = \mathcal{O}(\text{len})$ is easily shown to be an upper bound. In the other case, if no rotation is forced, the debits of the first len elements in f have to be discharged and the list-drop also runs in $\mathcal{O}(\text{len})$, and finally, to reestablish the debit invariant, it is necessary to discharge $(c+1) \cdot \text{len}$ debits on the front list (due to the index shift) and $c \cdot \text{len}$ debits on the rear list, all of which establishes the amortised runtime of $\mathcal{O}(\text{len})$. The argument for the case of a rotation being performed is similar to the ones given above, observing that before the drop, $c(|\text{f}| - x) + 1 - |\text{r}|$ for some $x < \text{len}$ and will not be repeated here.

Note that this establishes an amortised runtime complexity of $\mathcal{O}(\min(\text{len}, n - \text{len}))$ not only for drop, but by symmetry also for take. Then, of course, splitAt can also be implemented in $\mathcal{O}(\min(\text{len}, n - \text{len}))$, omitting the formal argument.

```
splitAt i d@(D lenf f r lenr)
  | i ≤ 0 = (empty, d)
  | i ≤ lenf = let (f',f'') = L.splitAt i f
               in (makeD i f' [] 0, makeD (lenf - i) f'' r lenr)
  | i ≤ lenf+lenr = let i' = i - lenf
                        (r', r'') = L.splitAt (lenr - i') r
                    in (makeD lenf f r'' i', makeD 0 [] r' (lenr - i'))
  | otherwise = (d, empty)
```

The remaining functions are all implemented using defaults.

concatenating multiple deques is performed by right-associatively appending them using foldr.

```
concat = concatUsingFoldr
concatMap = concatMapUsingFoldr
```

The foldr runs in $\mathcal{O}(n)$ plus the time needed for the appends, which is $\mathcal{O}(m)$ with m being the length of the resulting deque, plus the time needed for the map for concatMap, giving a total runtime of $\mathcal{O}(n + m)$.

As reverse runs in $\mathcal{O}(1)$, reverseOnto can be reduced to a reverse and an append, running in $\mathcal{O}(\min(n_1, n_2))$ like append.

```
reverseOnto = reverseOntoUsingReverse
```

As the size is explicitly stored, we can perform inBounds in constant time.

```
inBounds = inBoundsUsingSize
```

Using drop and take, it is easy to implement subseq. The total amortised time complexity is that of drop plus that of take, i.e. $\mathcal{O}(i + j)$ (for subseq i j).

```
subseq = subseqDefault
```

The *While operations traverse the deque by repeated application of lview. Per accepted element, this takes constant amortised time for lview, and cons if applicable, plus the time required for evaluation of the predicate, where at most n elements can be accepted.

```
takeWhile = takeWhileUsingLview
dropWhile = dropWhileUsingLview
splitWhile = splitWhileUsingLview
```

The zip/unzip-operations all operate by conversion to lists internally.

```
zip = zipUsingLists
zip3 = zip3UsingLists
zipWith = zipWithUsingLists
zipWith3 = zipWith3UsingLists
unzip = unzipUsingLists
unzip3 = unzip3UsingLists
unzipWith = unzipWithUsingLists
unzipWith3 = unzipWith3UsingLists
```

Like for the SimpleDeque, the reverse of the longer deque for a zip is only forced if its front list is shorter the shorter deque, which bounds the length of the longer deque to $(c+1)n_1 = \mathcal{O}(n_1)$. But up to the end of the front list, toList is incremental, so that again the amortised runtime complexity of the zips is $\mathcal{O}(\min(n_1, n_2))$.

Equality testing can be done a little more clever than usual by first comparing the sizes of the two involved deques, which takes constant time. The worst-case time is still linear in the length, of course.

```
instance Eq a ⇒ Eq (Seq a) where
  q1 == q2 =
    (size q1 == size q2) && (toList q1 == toList q2)
```

To generate an arbitrary BankersDeque, it is sufficient to generate arbitrary front and rear lists and then reduce the length of the longer to establish the invariant, if necessary. As the lists are of arbitrary length and accepted as they are if they fulfill the invariant, obviously any allowed BankersDeque can be generated this way.

```
instance Arbitrary a ⇒ Arbitrary (Seq a) where
  arbitrary =
    do f ← arbitrary
       r ← arbitrary
       return (let lenf  = L.size f
                   lenr  = L.size r
                   lenf' = min lenf (c*lenr+1)
                   lenr' = min lenr (c*lenf+1)
                in D lenf' (L.take lenf' f) (L.take lenr' r) lenr')
```

Generating an arbitrary BankersDeque with given elements makes it necessary to split the input list at a position such that the two parts fulfil the invariant. With $|\mathrm{xs}| = |\mathrm{f}| + |\mathrm{r}|$ we get from the invariant that

$$\begin{aligned} |\mathrm{r}| &\leq c|\mathrm{f}|+1 \\ |\mathrm{xs}|-|\mathrm{f}| &\leq c|\mathrm{f}|+1 \\ |\mathrm{xs}|-1 &\leq (c+1)|\mathrm{f}| \\ \left\lceil \frac{|\mathrm{xs}|-1}{c+1} \right\rceil = \left\lfloor \frac{|\mathrm{xs}|+c-1}{c+1} \right\rfloor &\leq |\mathrm{f}| \end{aligned}$$

and

$$\begin{aligned} |\mathrm{f}| &\leq c|\mathrm{r}|+1 \\ |\mathrm{f}| &\leq c(|\mathrm{xs}|-|\mathrm{f}|)+1 \\ (c+1)|\mathrm{f}| &\leq c|\mathrm{xs}|+1 \\ |\mathrm{f}| &\leq \left\lfloor \frac{c|\mathrm{xs}|+1}{c+1} \right\rfloor . \end{aligned}$$

So we pick a random value for $|\mathrm{f}|$ in this range and select front and rear list accordingly. As $|\mathrm{f}|$ may be anywhere in the allowed range, any allowed BankersDeque of the desired length with the given elements can be generated.

```
instance CheckSeq.CheckableSequence Seq where
  arbitraryWith xs = do let len = S.size xs
                        lenf ← choose ((len+c-1) `div` (c+1),
                                       (c*len + 1) `div` (c+1))
                        let lenr = len-lenf
                        let (f, r') = S.splitAt lenf xs
                        return (D lenf f (S.reverse r') lenr)
```

3.4 CatenableList

This design for a sequence that supports catenation in constant amortised time while also preserving constant time-bounds for the other basic list operations was proposed by Okasaki [Oka95a, Oka98] as a simpler alternative to data structures as proposed by Kaplan and Tarjan [KT95], that would, however, achieve worst-case time-bounds. The elements are stored in a multi-way tree, in pre-order left-to-right ordering. To store the children of a node, a queue, the `BankersQueue` in this case, is used.

```
data Seq a = E | C a (Q.Seq (Seq a))
```

The only structural invariant is that child-nodes can never be `E`. (If such a case would occur, the respective node is removed from the queue of children.)

```
invariant :: Seq a → Bool
invariant E = True
invariant xs = noE xs
  where noE E = False
        noE (C _ q) = Q.foldr (&&) True (Q.map noE q)
```

Amortised runtime analysis is carried out using the banker's method where $d_t(i)$ denotes the number of debits on node i in tree t and $D_t(i) = \sum_{j=0}^{i} d_t(i)$ the accumulated debits on nodes 0 through i. D_t is used as a short-hand for $D_t(|t| - 1)$, i.e. all debits on tree t.

Two debit invariants are maintained. First, the number of debits on each individual node must not exceed the node's degree. Note that the sum of all degrees in a tree is one less than the number of nodes and therefore $D_t < |t|$. Second, we require $D_t(i) \leq i + \mathit{depth}_t(i)$ and refer to this as left-linear debit invariant. As $d_t(0) = D_t(0) \leq 0 + 0 = 0$, this ensures that the root is free of debits.

Using these invariants, Okasaki has shown constant amortised time for `append` and `ltail`. We will repeat his arguments and analyse the remaining operations.

As usual, the two basic constructors are trivial constant time implementations.

```
empty = E
single x = C x Q.empty
```

Concatenating two `CatenableList`s is easily performed by adding the second one as last child to the root of the first.

```
append xs E = xs
append E ys = ys
append (C x q) ys = C x (Q.snoc q ys)
```

As `snoc` runs in constant amortised time, we can discharge the debit it would create immediately, and by observing that no node's degree decreases, we only need to verify that the left-linear debit invariant is preserved. For all nodes $i < |\mathrm{xs}|$, this is trivial because their index and depth remain the same and no new debits are created on them. For the nodes of the second list, their indexes are increased by $|\mathrm{xs}|$, their depth is increased by one and the accumulated debits are increased by the debits on xs, so we get

$$\begin{array}{rcl} D_t(|\mathrm{xs}|+i) & = & D_{\mathrm{xs}} + D_{\mathrm{ys}}(i) \\ & < & |\mathrm{xs}| + i + \mathit{depth}_{\mathrm{ys}}(i) \\ & = & |\mathrm{xs}| + i + \mathit{depth}_t(|\mathrm{xs}|+i) - 1 \\ & < & (|\mathrm{xs}|+i) + \mathit{depth}_t(|\mathrm{xs}|+i), \end{array}$$

where t denotes the resulting tree. Thus the left-linear debit invariant is indeed preserved and we have established constant amortised runtime for `append`.

Accessing the head element is easy and can obviously be performed in constant time as the root node is guaranteed to be free of debits. Removing it is a little more difficult, however. The solution is to `append` all children of the root node together, as depicted in figure 3.1. As

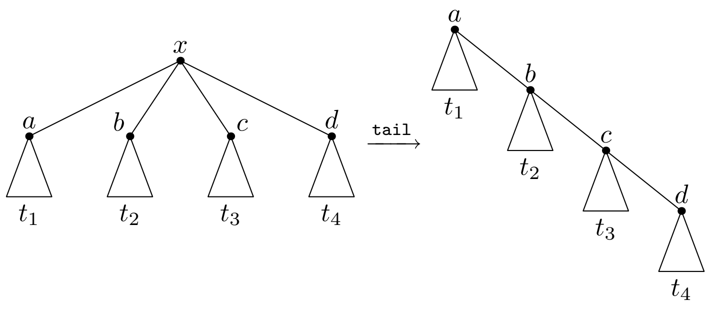


Figure 3.1: `tail` operation for `CatenableList`

`append` is commutative, the order in which this is performed does not influence the result. But to exploit lazy evaluation, it is most beneficial to do it right-associatively as the result will then be incremental when traversed from left to right.

```
lview E = Nothing2
lview (C x q) = Just2 x (Q.foldr append empty q)

lhead E = error "CatenableList.lhead:␣empty␣sequence"
lhead (C x _) = x

ltail E = E
ltail (C _ q) = Q.foldr append empty q
```

This way, one additional debit is placed on all former children of the root node and their degrees are increased by one, with the exception of the right-most child, so we immediately discharge the debit placed on it. Again, we only need to verify that the left-linear debit invariant is preserved. Now let `xs' = tail xs` and the ith node of `xs` reside in the jth subtree t_j. Now consider what happens to node i when `tail` is performed: its index is decreased to $i-1$, its depth in the tree is increased by $j-1$ and the accumulated debits up to it increase by j. Hence we get

$$\begin{array}{rcl} D_{\mathrm{xs}'}(i-1) & = & D_{\mathrm{xs}}(i) + j \\ & \leq & \mathit{depth}_{\mathrm{xs}}(i) + i + j \\ & = & \mathit{depth}_{\mathrm{xs}'}(i-1) - (j-1) + i + j \\ & = & \mathit{depth}_{\mathrm{xs}'}(i-1) + (i-1) + 2 \end{array}$$

and it is sufficient to discharge $2 = \mathcal{O}(1)$ more debits to restore the left-linear debit invariant.

To access or delete the right-most element, it is necessary to traverse the right spine. For the deletion, the construction of the resulting nodes becomes a little clumsy to avoid having empty leaves in the tree. If this would occur, instead the respective entry has to be removed from the queue of children.

```
rview E = Nothing2
rview (C x q)
  | Q.null q = Just2 E x
  | otherwise = let Just2 q' xs  = Q.rview q
                    Just2 xs' x' = rview xs
                in if null xs' then Just2 (C x q') x'
                   else Just2 (C x (Q.snoc q' xs')) x'

rhead E = error "CatenableList.rhead:␣empty␣sequence"
rhead (C x q) = if Q.null q then x else rhead (Q.rhead q)

rtail E = E
rtail (C x q)
  | Q.null q = E
  | otherwise = let Just2 q' xs = Q.rview q
                    xs'         = rtail xs
                in if null xs' then C x q' else C x (Q.snoc q' xs')
```

Accessing the right-most child of a node i runs in $\mathcal{O}(degree(i))$, so that traversing the right-spine takes up to $\sum_j degree(j) = \mathcal{O}(|\text{xs}|)$ (for nodes j on the right spine) steps, in addition to the up to $|\text{xs}|$ debits that have to be discharged, yielding linear amortised runtime.

Determining whether a list is empty is easily done in constant time by pattern matching on the root node.

```
null E = True
null _ = False
```

Determining the size requires a complete traversal, resulting in linear runtime complexity.

```
size E       = 0
size (C x q) = Q.foldr (+) 1 (Q.map size q)
```

The `map` and `fold` family of operations is implemented by using the respective queue operations to recurse down the tree. Only `foldr1` is excepted from this as the right-most element cannot easily be determined.

```
map _ E = E
map f (C x q) = C (f x) (Q.map (map f) q)

foldr _ e E       = e
foldr f e (C x q) = f x (Q.foldr (flip (foldr f)) e q)

foldl _ e E       = e
foldl f e (C x q) = (Q.foldl (foldl f) (f e x) q)
```

```
foldl1 _ E       = error "CatenableList.foldl1:␣empty␣sequence"
foldl1 f (C x q) = (Q.foldl (foldl f) x q)
```

As usual, the runtime is mainly determined by applying `f` n times.

Both `cons` and `snoc` simply `append` a singleton list, therefore requiring constant amortised time.

```
cons x s = append (single x) s
snoc = snocUsingAppend
```

Conversion from a list can be done by consecutive `cons`, which will result in a unary tree, i.e. a list, where all but the root node can be suspended without violating the debit invariants, thereby yielding constant amortised time. The same holds for `copy` and `tabulate` (assuming constant runtime for the function passed to the latter).

```
fromList = fromListUsingCons
copy = copyUsingLists
tabulate = tabulateUsingCons
```

Conversion to a list requires all debits to be discharged and the complete tree to be traversed, taking linear runtime. But it should be noted that the operation is incremental.

```
toList = toListUsingFoldr
```

Concatenating several lists is very similar to what happens in `ltail`.

```
concat = concatUsingFoldr
```

There is, however, one major difference: The concatenated lists are not guaranteed to be non-empty, but the empty ones will not remain in the resulting list, which may cause the depth increase to be less than what was stated in the analysis of `ltail`. In fact, if e is the number of empty lists, then the depth is reduced by e, so that e additional debits have to be discharged to reestablish the left-linear debit invariant, resulting in an amortised runtime complexity of $\mathcal{O}(1 + e)$.

For `concatMap`, this argument could be adapted, but still, the cost for the application of the function argument has to be paid, resulting in an amortised runtime complexity that is linear in the length of the input list (for a constant time function argument), but independent of the length of the result.

```
concatMap = concatMapUsingFoldr
```

Reversing a list obviously takes $\mathcal{O}(n)$, as yet alone accessing the last element does, and this cost cannot be left as debits, as it occurs on the first node of the result.

```
reverse = reverseUsingReverseOnto
reverseOnto = reverseOntoUsingFoldl
```

As mentioned above, `foldr1` cannot be easily implemented directly. Therefor, it uses the list representation internally, as do the `reduce` operations.

```
foldr1 = foldr1UsingLists
reducer = reducerUsingReduce1
reducel = reducelUsingReduce1
reduce1 = reduce1UsingLists
```

The different forms of sublist extraction all use the constant amortised time `ltail`/`lview` operations internally and therefore run in time linear to their arguments.

```
take = takeUsingLview
drop = dropUsingLtail
splitAt = splitAtUsingLview
subseq = subseqDefault
```

The `filter` and `partition` operations use `foldr` to traverse the list and `cons` to build the appropriate result(s).

```
filter = filterUsingFoldr
partition = partitionUsingFoldr
```

Likewise, the `*While` family of operations uses `lview` to traverse the list as long as necessary.

```
takeWhile = takeWhileUsingLview
dropWhile = dropWhileUsingLview
splitWhile = splitWhileUsingLview
```

Checking whether a given index is inside the list or looking up the respective element is performed by `drop`, thus running in $\mathcal{O}(i)$.

```
inBounds = inBoundsUsingDrop
lookup = lookupUsingDrop
lookupM = lookupMUsingDrop
lookupWithDefault = lookupWithDefaultUsingDrop
```

Changing an element at a given index uses `splitAt` to reach and change the desired element (taking linear time) and then `append`s the two parts together again.

```
update = updateUsingSplitAt
adjust = adjustUsingSplitAt
```

The remaining operations all use the list representation internally and therefore obviously run in $\mathcal{O}(n)$, as `toList`, `fromList` and the respective list operations do. For the `zip` family, the shortest list determines the runtime as `toList` is incremental.

```
mapWithIndex = mapWithIndexUsingLists
foldrWithIndex = foldrWithIndexUsingLists
foldlWithIndex = foldlWithIndexUsingLists
zip = zipUsingLists
zip3 = zip3UsingLists
zipWith = zipWithUsingLists
zipWith3 = zipWith3UsingLists
unzip = unzipUsingLists
unzip3 = unzip3UsingLists
unzipWith = unzipWithUsingLists
unzipWith3 = unzipWith3UsingLists
```

Generating an arbitrary multi-way tree becomes a little messy, so we just generate an arbitrary list and then use `arbitraryWith` to avoid duplicating work.

```
instance Arbitrary a ⇒ Arbitrary (Seq a) where
  arbitrary = arbitrary ≫= CheckSeq.arbitraryWith
```

Generating an arbitrary CatenableList for a given list of elements is a little involved. If the list is empty, the result trivially is the empty CatenableList, of course. Otherwise, the head of the list becomes the element in the root, and the tail is partitioned into an arbitrary number of parts for which in turn arbitrary CatenableLists are generated which are stored in the queue of child nodes. The partitioning is performed by recursing over the list from right to left, every time taking at least one and at most all elements to form the right-most child-node. This will result in trees that are unbalanced towards the right on average. However, all forms of trees are possible: Both the number of children and the number of elements per child are completely arbitrary.

```
instance CheckSeq.CheckableSequence Seq where
  arbitraryWith [] = return empty
  arbitraryWith (x:xs) = liftM (C x) (q xs)
    where q [] = return Q.empty
          q xs = do n ← choose (0, L.size xs - 1)
                    let (xs', xs'') = L.splitAt n xs
                    ls ← q xs'
                    l ← CheckSeq.arbitraryWith xs''
                    return (Q.snoc ls l)
```

3.5 CatenableDeque

The catenable deques (supporting the standard deque operations and append in constant amortised time) as described in [Oka97a, Oka98] are by far the most complicated sequence data structure implemented in the course of this project. It requires an implementation of normal deques (i.e. with linear time append), for which we will use BankersDeque as it is at present the only deque implementation that also features persistence. Furthermore, its append runs in time linear to the length of the shorter of the two arguments, a property that comes in handy later on.

A catenable deque (c-deque) can have to forms, either Shallow or Deep.

```
import qualified BankersDeque as D

data Seq a = Shallow !(D.Seq a)
           | Deep !(D.Seq a)
                  (Seq (CmpdElem a))
                  !(D.Seq a)
                  (Seq (CmpdElem a))
                  !(D.Seq a)
```

In its Shallow form, it merely is a wrapper for the normal deque. The more interesting Deep form represents a quintuple (f, a, m, b, r), where f, m and r are normal deques again and a and b are c-deques of compound elements. f and r must always contain at least three elements, m at least two; should the length of a c-deque in Deep form be reduced to below eight, it will be converted back to Shallow form.

The compound elements also come in two flavours, Simple or Cmpd.

```
data CmpdElem a = Simple !(D.Seq a)
                | Cmpd !(D.Seq a) (Seq (CmpdElem a)) !(D.Seq a) deriving Show
```

The `Simple` compound element is again just a wrapper for a normal deque, which is this time required to contain at least two elements, while the `Cmpd` form represents a triple (f, c, r), where f and r are normal deques, again with at least two elements, and c is a c-deque of compound elements.

```
invariant :: Seq a → Bool
invariant (Shallow _) = True
invariant (Deep f a m b r) = D.size f ≥ 3
                             && invariant a
                             && foldr (λx e → cmpdInvariant x && e) True a
                             && D.size m ≥ 2
                             && invariant b
                             && foldr (λx e → cmpdInvariant x && e) True b
                             && D.size r ≥ 3

cmpdInvariant :: CmpdElem a → Bool
cmpdInvariant (Simple d) = D.size d ≥ 2
cmpdInvariant (Cmpd f c r) = D.size f ≥ 2
                             && invariant c
                             && foldr (λx e → cmpdInvariant x && e) True c
                             && D.size r ≥2
```

Note that the strictness flags used for all occurrences of normal deques of course do not interfere with the laziness used inside them. The strictness only avoids the overhead that would occur for the creation of the outmost suspensions, which would not yield any gain, as all operations suspended in this way run in $\mathcal{O}(1)$ amortised time anyway.

The amortised runtime analysis will again be carried out using the banker's method. As only the a and b fields of the `Deep` c-deque and the c field of the `Cmpd` can carry suspensions, debits are only placed on these. The debit invariant introduced by Okasaki for the `Cmpd` is that at most four debits may be placed on the c field of `Cmpd` and between zero and five on the a and b fields, as defined by the following rules:

- a and b have a base allowance of zero debits.
- If f contains more than three elements, then the allowance for a is increased by four debits and that of b by one.
- Likewise, if r contains more than three elements, then the allowance for b is increased by four debits and that of a by one.

With these, Okasaki showed constant amortised runtime for `append`, `cons`, `snoc`, `ltail` and `rtail`. We will extent his ideas to the remaining operations, as well as outline his arguments.

For operations inside the `Shallow` form, no additional analysis will be necessary as they are directly mapped to the respective operations of the normal deque.

Creating an empty or singleton c-deque simply creates a `Shallow` c-deque carrying the appropriate deque (`empty` is directly taken from [Oka98]). The runtime is obviously constant.

```
empty = Shallow D.empty
single x = Shallow (D.single x)
```

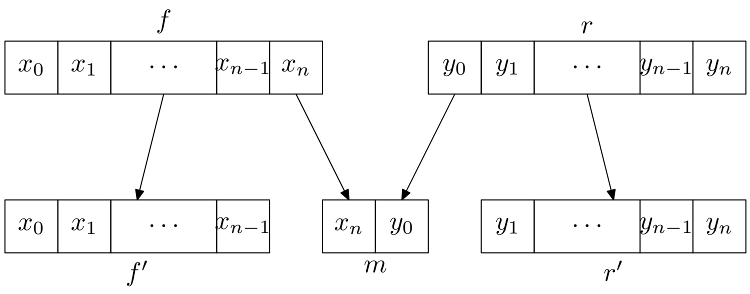
Adding a single element, either to the front or rear end, just has to add the element to the respective internal deque. (Both operations are described in [Oka97a], and [Oka98] contains Haskell code for `cons`.)

```
cons x (Shallow d) = Shallow (D.cons x d)
cons x (Deep f a m b r) = Deep (D.cons x f) a m b r

snoc (Shallow d) x = Shallow (D.snoc d x)
snoc (Deep f a m b r) x = Deep f a m b (D.snoc r x)
```

As both `cons` and `snoc` for the `BankersDeque` run in constant amortised time and obviously the debit invariant is preserved, both operations run in $\mathcal{O}(1)$ amortised.

The `append` operation has to distinguish the four major cases whether both c-deques are `Shallow`, the first or the second is `Deep`, or both are `Deep`, where of course the second and third case are symmetric. If both c-deques are `Shallow` and at least one of them contains less than four elements, we can call the `append` operation of the `BankersDeque` directly and create a new `Shallow` c-deque. If they both contain four or more elements, a `Deep` c-deque is built with the a and b components empty, the m component containing the last element of the first c-deque and the first element of the of the second c-deque, and the f and r components set to the remainders of the two internal deques, respectively. (See figure 3.2.) This way of dividing two



f
r
x_0 x_1 $\cdots$ x_{n-1} x_n
y_0 y_1 $\cdots$ y_{n-1} y_n
x_0 x_1 $\cdots$ x_{n-1}
x_n y_0
y_1 $\cdots$ y_{n-1} y_n
f'
m
r'

Figure 3.2: `share` helper function for `CatenableDeque`

deques into three occurs several times, so we introduce a helper function `share` to take care of this. (This version is adapted from the one given in [Oka98] and modified to make use of the `view` operations. Likewise, the `append` implementations are modified to exploit that the `append` of the `BankersDeque` runs in time linear to the length of its shorter argument, making it unnecessary to provide two different, specialised `append`s.)

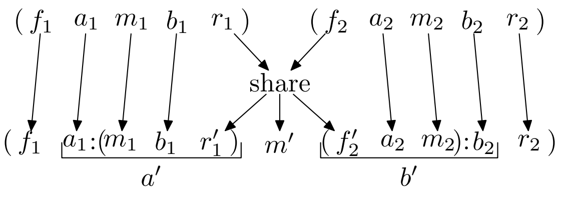
```
share :: D.Seq a → D.Seq a → (D.Seq a, D.Seq a, D.Seq a)
share f r = let Just2 fi fl = D.rview f
                Just2 rh rt = D.lview r
            in (fi, D.cons fl (D.single rh), rt)

append (Shallow d1) (Shallow d2)
  | (D.size d1<4) || (D.size d2<4) = Shallow (D.append d1 d2)
  | otherwise = Deep f empty m empty r
                where (f, m, r) = share d1 d2
```

Appending a `Shallow` and a `Deep` c-deque again distinguishes whether the `Shallow` c-deque has less than four elements or not. If so, its internal deque is appended to the f (or r, respectively) component of the `Deep` c-deque. Otherwise, the internal deque of the `Shallow` c-deque becomes the new f (or r, respectively) and the old f (r) is added to the a (b) component as a `Simple` compound element.

```
append (Shallow d) (Deep f a m b r)
  | D.size d<4 = Deep (D.append d f) a m b r
  | otherwise  = Deep d (cons (Simple f) a) m b r
append (Deep f a m b r) (Shallow d)
  | D.size d<4 = Deep f a m b (D.append r d)
  | otherwise  = Deep f a m (snoc b (Simple r)) d
```

The last case, appending two `Deep` c-deques, is the most complicated one. Instead of translating the very concise Haskell code into English that is not necessarily any easier to understand, figure 3.3 tries to visualise what is done (note that the : denotes both `cons` and `snoc`, respectively).



$(f_1\ a_1\ m_1\ b_1\ r_1)$
$(f_2\ a_2\ m_2\ b_2\ r_2)$
share
$(f_1\ a_1{:}(m_1\ b_1\ r_1')\ m'\ (f_2'\ a_2\ m_2){:}b_2\ r_2)$
a'
b'

Figure 3.3: `append` for two `Deep` `CatenableDeque`s

```
append (Deep f1 a1 m1 b1 r1) (Deep f2 a2 m2 b2 r2)
  = let (r1', m, f2') = share r1 f2
        a1' = snoc a1 (Cmpd m1 b1 r1')
        b2' = cons (Cmpd f2' a2 m2) b2
    in Deep f1 a1' m b2' r2
```

All functions called by `append` have constant amortised runtime: For `share`, this follows directly from the constant amortised runtime of the called deque functions, for `cons` and `snoc`, this was established above, and for the deque-`append`, at least one of its arguments has three or less arguments, bounding its runtime. This leaves the debits for the cases involving `Deep` c-deques to be analysed. However, as the debits on a and b are at most five, we can discharge all of them (at most 20), plus those two that may be created by the `cons` and `snoc`, which obviously is sufficient to preserve the debit invariant, and still get constant amortised runtime. (A more careful analysis shows that it is actually enough to discharge four debits [Oka97a]).

Accessing the head element is trivially done in constant amortised time using the respective deque operation.

```
lhead (Shallow d) = D.lhead d
lhead (Deep f _ _ _ _) = D.lhead f
```

Removing the head element requires much more thought. The `Shallow` form is quite easy, again; also for a `Deep` c-deque with f containing four or more elements, it is sufficient just to remove the head of f. Otherwise (i.e. f containing the minimum number of three elements), several cases have to distinguished:

- If a is non-empty, we can remove its head element to acquire a second deque that we `append` to the tail of f to get back to at least three elements. If the head element of a is a `Simple` compound element, the deque it carries is the one we need. For a `Cmpd`, its f component is the desired deque, but we need to take care of its c and r components as well. We `cons` the r component (encapsulated in a `Simple` compound element) to the tail of a and then append the result to c, which is a c-deque itself, to get the new a.
- If a is empty, but b is not, we append m to the tail of f to arrive at a resulting f with at least three elements and extract the head of b to obtain a new m. If the head of b is a `Simple` compound element, the deque it carries is our new m and we are done. If it is a `Cmpd`, its r component becomes the new m, while its f and c components are joined together (by encapsulating f in a `Simple` compound element and `cons`ing to c) to replace the formerly empty a.
- If both a and b are empty, tail of f and m are appended, the result is put in a `Shallow` c-deque, as is r, and these two c-deques are then `append`ed to form the desired result.

In the case of $|f| = 3$ and head of a being a `Cmpd` compound element, we immediately replace a's head after removing it. In that case, it is of course unnecessary to preserve the minimum length of a's f component during the `ltail`, so we define an auxiliary function that handles this `cons x (ltail cd)` a little more efficiently.

```
replacelhead :: a → Seq a → Seq a
replacelhead x (Shallow d) = Shallow (D.cons x (D.ltail d))
replacelhead x (Deep f a m b r) = Deep (D.cons x (D.ltail f)) a m b r
```

Putting all that together, we arrive at the rather lengthy implementation of `ltail` (almost literally as in [Oka98]).

```
ltail (Shallow d) = Shallow (D.ltail d)
ltail (Deep f a m b r)
  | D.size f>3 = Deep (D.ltail f) a m b r
  | not (null a) = case lhead a of
                      Simple d → Deep f' (ltail a) m b r
                        where f' = D.append (D.ltail f) d
                      Cmpd f' c' r' → Deep f'' a'' m b r
                        where f'' = D.append (D.ltail f) f'
                              a'' = append c' (replacelhead (Simple r') a)
  | not (null b) = case lhead b of
                      Simple d → Deep f' a d (ltail b) r
                        where f' = D.append (D.ltail f) m
                      Cmpd f' c' r' → Deep f'' a'' r' (ltail b) r
                        where f'' = D.append (D.ltail f) m
                              a'' = cons (Simple f') c'
  | otherwise = append (Shallow (D.append (D.ltail f) m)) (Shallow r)
```

In the case of a `Shallow` c-deque, `ltail` obviously runs in $\mathcal{O}(1)$ amortised. Otherwise, by inspection, all functions invoked inside `ltail` run in constant amortised time, with the possible exception of the recursion, which is suspended. So again, the debit analysis becomes the critical part. Due to the recursive structure, we will argue by debits that are passed to the surrounding suspension; at the out-most level, these are the debits that are actually to be discharged. We will establish that at most five debits are passed to the surrounding suspension, dealing with the different cases distinctly.

$|f| > 3$: No recursive call occurs, but it may be the case that $|f|$ drops to three, thereby reducing the debit allowance on a by four and on b by one, making it necessary to pass up to five debits to the surrounding suspension.

$|f| = 3$, a non-empty, head of a is `Simple`: If $|r| > 3$, a may carry one debit which we pass out to the surrounding suspension. As the f component of the resulting c-deque will have more than three elements, we may leave four or five (depending or r) debits of the five we inherit from the recursive call on a, so we have to pass out one debit now if $|r| = 3$, and hence always pass out at most one debit.

$|f| = 3$, a non-empty, head of a is `Cmpd`: Again, if $|r| > 3$ we have to pass one debit from a on to the surrounding suspension. But the debits associated with the new a are higher: The c component may carry up to four debits, the `append` produces up to four debits and the `replacelhead` produces one additional debit, summing up to nine debits. If $|r| = 3$ we have to pass five of them on, otherwise four are sufficient, so that at most five debits have to be passed on in total.

$|f| = 3$, a empty, b non-empty, head of b is `Simple`: If $|r| > 3$, we have to pass up to four debits from b on to the surrounding suspension, but can leave all the five debits generated by the recursive call to `ltail`, as afterwards also $|f| > 3$. Otherwise, b is guaranteed to be free of debits, but we have to pass four of the five generated debits out. Hence, at most four debits have to be passed out.

$|f| = 3$, a empty, b non-empty, head of b is `Cmpd`: For the debits on b the same argument holds, but we also have to considers the new a component. The c component of the compound element may carry up to four debits, and the `cons` generates one additional debit. If $|r| = 3$, only four debits are allowed on a, so we have to pass out one more debit, yielding a total of five debits to be passed on to the surrounding suspension.

$|f| = 3$, a and b empty: No recursive call occurs, but the call to `append` may result in up to four debits to be passed on.

So, indeed `ltail` runs in $\mathcal{O}(1)$ amortised.

For `lview`, we use `lhead` and `ltail` to come up with a straight-forward $\mathcal{O}(1)$ implementation.

```
lview cd | null cd = Nothing2
         | otherwise = Just2 (lhead cd) (ltail cd)
```

Exploiting symmetry, we can now define `rhead`, `rtail` and `rview` in a similar way, also yielding constant amortised runtime.

```
rhead (Shallow d) = D.rhead d
rhead (Deep _ _ _ _ r) = D.rhead r

replacerhead (Shallow d) x = Shallow (D.snoc (D.rtail d) x)
replacerhead (Deep f a m b r) x = Deep f a m b (D.snoc (D.rtail r) x)

rtail (Shallow d) = Shallow (D.rtail d)
rtail (Deep f a m b r)
```

```
    | D.size r>3 = Deep f a m b (D.rtail r)
    | not (null b) = case rhead b of
                       Simple d → Deep f a m (rtail b) r'
                         where r' = D.append d (D.rtail r)
                       Cmpd f' c' r' → Deep f a m b'' r''
                         where r'' = D.append r' (D.rtail r)
                               b'' = append (replacerhead b (Simple f')) c'
    | not (null a) = case rhead a of
                       Simple d → Deep f (rtail a) d b r'
                         where r' = D.append m (D.rtail r)
                       Cmpd f' c' r' → Deep f (rtail a) f' b'' r''
                         where r'' = D.append m (D.rtail r)
                               b'' = snoc c' (Simple r')
    | otherwise = append (Shallow f) (Shallow (D.append m (D.rtail r)))

  rview cd | null cd = Nothing2
           | otherwise = Just2 (rtail cd) (rhead cd)
```

As only a `Shallow` c-deque can be empty, the test whether a given c-deque is empty reduces to the respective test for normal deques (which runs in constant amortised time).

```
  null (Shallow d) = D.null d
  null _ = False
```

Determining the size of a c-deque is a little more involved for the `Deep` form.

```
  size (Shallow d) = D.size d
  size (Deep f a m b r) = D.size f + foldr size' 0 a + D.size m
                                   + foldr size' 0 b + D.size r
    where size' (Simple d) n = D.size d + n
          size' (Cmpd f c r) n = D.size f + foldr size' 0 c + D.size r + n
```

The deque `size` operation runs in constant time, and all deques are non-empty, so that these contribute at most linear runtime in total, as only $\mathcal{O}(n)$ applications of the deque `size` operation can be performed. As each (recursive) call to `size'` results in at least one call to `size`, the total number of these is obviously also linear in n. Furthermore, the amount of debits discharged per recursion is constant, thereby yielding amortised runtime $\mathcal{O}(n)$.

Reversing a `Shallow` `CatenableDeque` is carried out by simply reversing the deque used inside. For the `Deep` form, the order of the five components has to be reversed, the components themselves have to be reversed, and for the a and b components, the compound elements they contain have to be reversed.

```
  reverse (Shallow d) = Shallow (D.reverse d)
  reverse (Deep f a m b r)
    = Deep f' a' m' b' r'
        where f' = D.reverse r
              a' = reverse (map rev' b)
              m' = D.reverse m
              b' = reverse (map rev' a)
              r' = D.reverse f
              rev' (Simple d) = Simple (D.reverse d)
```

```
            rev' (Cmpd f c r) = Cmpd (D.reverse r) (reverse (map rev' c))
                                     (D.reverse f)
```

Observing that the deque `reverse` operation is $\mathcal{O}(1)$, with the same argument as for `size`, linear amortised runtime can be established.

Conversion of a list into a `CatenableDeque` is performed by just converting the list into a deque and then wrapping it up in the `Shallow` constructor, therefore taking linear amortised time like the respective deque function.

```
fromList xs = Shallow (D.fromList xs)
```

The `map` and `fold` family of operations is implemented by calls to the respective deque operations, taking special care of the compound elements. As the total number of debits is $\mathcal{O}(n)$, all of which have to be discharged, again the n-fold application of the function argument `g` determines the runtime.

```
map g (Shallow d) = Shallow (D.map g d)
map g (Deep f a m b r)
  = Deep f' a' m' b' r'
      where f' = D.map g f
            a' = map g' a
            m' = D.map g m
            b' = map g' b
            r' = D.map g r
            g' (Simple d) = Simple (D.map g d)
            g' (Cmpd f c r) = Cmpd (D.map g f) (map g' c) (D.map g r)

foldr g e (Shallow d) = D.foldr g e d
foldr g e (Deep f a m b r) = D.foldr g (foldr g' (D.foldr g (foldr g'
                                         (D.foldr g e r) b) m) a) f
  where g' (Simple d) e = D.foldr g e d
        g' (Cmpd f c r) e = D.foldr g (foldr g' (D.foldr g e r) c) f

foldr1 g (Shallow d) = D.foldr1 g d
foldr1 g (Deep f a m b r) = D.foldr g (foldr g' (D.foldr g (foldr g'
                                         (D.foldr1 g r) b) m) a) f
  where g' (Simple d) e = D.foldr g e d
        g' (Cmpd f c r) e = D.foldr g (foldr g' (D.foldr g e r) c) f

foldl g e (Shallow d) = D.foldl g e d
foldl g e (Deep f a m b r) = D.foldl g (foldl g' (D.foldl g (foldl g'
                                         (D.foldl g e f) a) m) b) r
  where g' e (Simple d) = D.foldl g e d
        g' e (Cmpd f c r) = D.foldl g (foldl g' (D.foldl g e f) c) r

foldl1 g (Shallow d) = D.foldl1 g d
foldl1 g (Deep f a m b r) = D.foldl g (foldl g' (D.foldl g (foldl g'
                                         (D.foldl1 g f) a) m) b) r
  where g' e (Simple d) = D.foldl g e d
        g' e (Cmpd f c r) = D.foldl g (foldl g' (D.foldl g e f) c) r
```

The `copy` and `tabulate` constructors again just wrap up the respective deque operations in `Shallow` constructors, preserving their constant amortised runtime.

```
copy n x = Shallow (D.copy n x)
tabulate n f = Shallow (D.tabulate n f)
```

While `drop` on a `Shallow` `CatenableDeque` again just calls the respective deque operation, we distinguish two cases for the `Deep` form. If $|f| - i \geq 3$, we can just `drop` i elements from f without violating the invariant. Otherwise, we `drop` all but three elements, call `ltail` to remove the next and then call `drop` recursively to remove the remaining elements.

```
drop i (Shallow d) = Shallow (D.drop i d)
drop i (Deep f a m b r)
  | D.size f - i ≥ 3 = Deep (D.drop i f) a m b r
  | otherwise = drop (i - D.size f + 2)
                     (ltail (Deep (D.drop (D.size f - 3) f) a m b r))
```

The non-recursing cases obviously take linear time, as so does the deque `drop`, and at most five debits have to be discharged in case the length of f drops to three. As both `ltail` as `D.drop (D.size f - 3) f` run in constant amortised time, and i is reduced by at least five in the recursive call, we get amortised runtime $\mathcal{O}(i)$ in total.

The case distinction for `take` is similar. If the `CatenableDeque` is `Shallow` or the index lies within f, we can just use the deque `take` and wrap the result up in a `Shallow` `CatenableDeque` again. Otherwise, we `append` f to the first $i - |f|$ elements of the `CatenableDeque` that remains after dropping $|f|$ elements.

```
take i (Shallow d) = Shallow (D.take i d)
take i d@(Deep f a m b r)
  | i ≤ D.size f = Shallow (D.take i f)
  | otherwise = append (Shallow f) (take (i-D.size f) (drop (D.size f) d))
```

The non-recursing cases again run in $\mathcal{O}(i)$ amortised. In the recursing case, the `drop` runs in constant amortised time, as a closer inspection reveals: It recurses only once, with $i = 2 = \mathcal{O}(1)$. As `append` also runs in constant amortised time and the i is reduced by at least $|f| \geq 3$ in the recursive call, we get total amortised runtime $\mathcal{O}(i)$.

Checking whether a given index is valid for the `Shallow` form is again performed using the respective deque operation. For the `Deep` form, we exploit the constant time `size` operation of the deque to first check whether the elements stored in the f, m and r components are already sufficient to establish that the index is valid. If not, we use `drop`, resulting in amortised runtime of $\mathcal{O}(i)$.

```
inBounds _ i | i<0 = False
inBounds (Shallow d) i = D.inBounds d i
inBounds (Deep f a m b r) i
  | i<(D.size f + D.size m + D.size r) = True
  | otherwise = inBoundsUsingDrop (Deep f a m b r) i
```

Concatenating multiple `CatenableDeque` can conveniently be performed by repeated application of the constant time `append`, yielding an amortised runtime linear in the number of `CatenableDeque`s to be concatenated. For `concatMap`, the time required for the given function has to be added, of course.

```
concat = concatUsingFoldr
concatMap = concatMapUsingFoldr
```

The `reverseOnto` operation uses `reverse` and `append`, resulting in linear amortised runtime.

```
reverseOnto = reverseOntoUsingReverse
```

Conversion to a list uses `foldr` to `cons` the result list together. This results in a list that can be evaluated incrementally, with a total runtime of $\mathcal{O}(n)$.

```
toList = toListUsingFoldr
```

As it is quite complex to divide a `CatenableDeque` evenly, the `reduce` family of operations uses a list representation internally, giving again runtime that is determined by the n-fold application of the function argument.

```
reducer = reducerUsingReduce1
reducel = reducelUsingReduce1
reduce1 = reduce1UsingLists
```

Both `splitAt` and `subseq` call `take` and `drop` internally, thus having amortised runtime $\mathcal{O}(i)$ and $\mathcal{O}(i+j)$, respectively.

```
splitAt = splitAtDefault
subseq = subseqDefault
```

The `lookup` family of operations just `drop`s the first i elements to access the requested index, resulting in an amortised runtime of $\mathcal{O}(i)$.

```
lookup = lookupUsingDrop
lookupM = lookupMUsingDrop
lookupWithDefault = lookupWithDefaultUsingDrop
```

Changing an element using `update` or `adjust` is performed by splitting the `CatenableDeque` at the given index, modifying the respective element and `append`ing the two parts together again. In addition to the potentially expensive application of the function argument in `adjust`, the `splitAt` with its amortised runtime $\mathcal{O}(i)$ determines the total runtime.

```
update = updateUsingSplitAt
adjust = adjustUsingSplitAt
```

The runtime of the `*WithIndex` operations, which use lists internally, is again dominated by the application of `f` n times.

```
mapWithIndex = mapWithIndexUsingLists
foldrWithIndex = foldrWithIndexUsingLists
foldlWithIndex = foldlWithIndexUsingLists
```

`filter` and `partition` use `foldr` to traverse the `CatenableDeque` and build the result(s), as usual having time-complexity that is determined by the n-fold application of the function argument `f`.

```
filter = filterUsingFoldr
partition = partitionUsingFoldr
```

The same holds for the `*While` operation, which use `lview` to traverse the `CatenableDeque` as far as required.

```
takeWhile = takeWhileUsingLview
dropWhile = dropWhileUsingLview
splitWhile = splitWhileUsingLview
```

As `toList` is incremental, the `zip` family of operations, using list representations internally, has time linear in the length of the shortest `CatenableDeque` involved.

```
zip = zipUsingLists
zip3 = zip3UsingLists
zipWith = zipWithUsingLists
zipWith3 = zipWith3UsingLists
```

For the `unzip` operations, the traversal is performed using `foldr`, and the resulting runtime once again linear or determined by the function argument, respectively.

```
unzip = unzipUsingFoldr
unzip3 = unzip3UsingFoldr
unzipWith = unzipWithUsingFoldr
unzipWith3 = unzipWith3UsingFoldr
```

Generation of arbitrary `CatenableDeque`s is a little more involved than for the other implementations, as we need to implement the `Arbitrary` interface for two data types.

Observing that the structural invariant demands several deques to have a certain minimum size, we start of by defining a helper function to produce an arbitrary `BankersDeque`s with a given minimum size. We do this by simply generating a deque of arbitrary size and then adding n more elements to it to ensure a minimum size of n.

```
minLengthDeque :: (Arbitrary a) ⇒ Int → Gen (D.Seq a)
minLengthDeque 0 = arbitrary
minLengthDeque n = do x ← arbitrary
                      xs ← minLengthDeque (n-1)
                      return (D.cons x xs)
```

With this helper function, we can easily generate an arbitrary compound element that fulfils the invariant. As obviously no other restrictions as to what is generated then the minimum lengths apply, any allowed compound element can be produced. It should be noted that even for very small generation sizes, the `Cmpd` case may be entered resulting in a `CmpdElem` with at least four elements. We allow this to get more interesting test cases, i.e. non-shallow instances.

```
instance Arbitrary a ⇒ Arbitrary (CmpdElem a) where
  arbitrary = oneof [ simple, sized cmpd ]
    where simple = liftM Simple (minLengthDeque 2)
          cmpd size = resize (size `div` 3) $
                        liftM3 Cmpd (minLengthDeque 2) arbitrary (minLengthDeque 2)
```

Generation of an arbitrary `CatenableDeque` works in a very similar way, except that we restrict generation to the `Shallow` variant for small sizes to avoid infinite recursion. Again, it easy to see that any allowed `CatenableDeque` can be generated.

```
instance Arbitrary a ⇒ Arbitrary (Seq a) where
```

```
    arbitrary = oneof [ shallow, sized deep ]
      where shallow = liftM Shallow arbitrary
            deep size | size<8    = shallow
                      | otherwise = resize (size ‘div‘ 5) $
                                    liftM5 Deep (minLengthDeque 3)
                                                arbitrary
                                                (minLengthDeque 2)
                                                arbitrary
                                                (minLengthDeque 3)
```

Generation of a `CatenableDeque` with given contents is trickier as we cannot arbitrarily add elements to the deques to provide the required minimum size. Instead, we have to divide the given elements in a legal way. We introduce additional helper functions to accomplish this, starting with one that produces a list of numbers that add up to a given value. As an extra argument, it takes a list of lists of allowed values. The advantage of this compared to a list of minimum values, which would seem sufficient at first, is that it also allows the case where zero elements are allowed as well, but at least two otherwise. The domain lists have to in increasing order; this permits the function to just repeatedly remove the head element of a randomly chosen domain list until all head elements add up to the desired value. Before removal of the head element, it is ensured that this will not result in a too high value. Of course, this approach can get stuck if e.g. the sum has to be increased by only one, but all possible increases are at least two. However, in all applications that follow below, at least one domain list will only consist of increases by one.

```
  fillToSum :: [[Int]] → Int → Gen [Int]
  fillToSum xxs n = if s ≥ n then return (S.map S.lhead xxs)
                    else do i ← choose (0, length xxs -1)
                            fillToSum (S.adjust maybeTail i xxs) n
                    where s = sum (S.map S.lhead xxs)
                          maybeTail (x1:x2:xs) = if x2-x1 ≤ n-s then x2:xs
                                                 else x1:x2:xs
```

Secondly, we define a function that splits a list into parts, where the lengths of the parts can be restricted with the same kind of domain lists as explained above. The function is straight-forward by using `fillToSum`.

```
  splitToSizes :: [[Int]] → [a] → Gen [[a]]
  splitToSizes is xs = liftM (splitMulti xs) (fillToSum is (S.size xs))
    where splitMulti [] [] = []
          splitMulti xs (i:is) = let (xs’, xs’’) = S.splitAt i xs
                                     xxs = splitMulti xs’’ is
                                 in (xs’:xxs)
```

Finally we can turn to the main `arbitraryWith` function. First, we examine the length of the input list: if it is empty, so is the result; if it contains less than eight elements, the result has to be `Shallow`; otherwise, both `Shallow` and `Deep` form are possible.

```
  instance CheckSeq.CheckableSequence Seq where
    arbitraryWith [] = return empty
    arbitraryWith xs
      | S.size xs < 8 = shallow
      | otherwise = oneof [ shallow, deep ]
```

While generation of the `Shallow` form is trivial, we need to use the helper function introduced above to split the input list into suitable parts. The deque parts *f*, *m* and *r* have simple minimum sizes, while *a* and *b* may be empty, but cannot contain only one element, because a `CmpdElem` has to contain at least two elements. To generate *a* and *b*, we first generate a list of compound elements with `cmpdList` (see below), and then call `arbitraryWith` on this recursively.

```
  where shallow = return (Shallow (D.fromList xs))
        deep = do [f', a', m', b', r'] ← splitToSizes [[3..],
                                                         (0:[2..]),
                                                         [2..],
                                                         (0:[2..]),
                                                         [3..]] xs
                   a ← (cmpdList a' ≫= CheckSeq.arbitraryWith)
                   b ← (cmpdList b' ≫= CheckSeq.arbitraryWith)
                   return (Deep (D.fromList f')
                                a
                                (D.fromList m')
                                b
                                (D.fromList r'))
```

The list of compound elements is generated by first choosing its length, then dividing the input elements accordingly, and finally wrapping them up in `CmpdElem`s.

```
cmpdList :: [a] → Gen [CmpdElem a]
cmpdList [] = return []
cmpdList xs = do n ← choose (1, S.size xs `div` 2)
                 xxs ← splitToSizes (S.copy n [2..]) xs
                 mapM arbitraryCmpdWith xxs
```

Generation of an arbitrary compound element for given contents is similar to generation of a `CatenableDeque`. If there are less than four elements, it can only be `Simple`, otherwise both forms are allowed.

```
arbitraryCmpdWith :: [a] → Gen (CmpdElem a)
arbitraryCmpdWith xs
  | S.size xs < 4 = simple xs
  | otherwise = oneof [ simple xs, cmpd xs ]
```

Again, the `simple` case is indeed simple, while the `cmpd` case works in a way similar to the `deep` case above.

```
simple xs = return (Simple (D.fromList xs))
cmpd xs = do [f', c', r'] ← splitToSizes [[2..],
                                          (0:[2..]),
                                          [2..]] xs
             c ← (cmpdList c' ≫= CheckSeq.arbitraryWith)
             return (Cmpd (D.fromList f') c (D.fromList r'))
```

As the `CatenableDeque` is independent of the internals of the `BankersDeque`, which is moreover expected to correct (or at least checked by itself), `fromList` can safely be used to produce the deques instead of `arbitraryWith`.

Chapter 4

Discussion

4.1 Related Work

While a lot of theoretical work about various functional data structures has been done, most of them have been implemented only outside any frameworks if at all. But especially the design of a library, not the details of implementations, are of special interest, to be able to spot possible design flaws in Edison. We will therefore compare Edison with imperative data structure libraries, namely the Java Collection Framework and the Standard Template Library of C++, after examining what has been done towards data structure libraries in functional settings.

4.1.1 Other Functional Data Structure Libraries

The only other library dedicated to functional data structures the author was able to dig up was *Baire* for the Caml language by Diego Olivier Fernandez Pons. Baire is loosely based on Edison and FGL[1] and furthermore contains a variety of higher-level data structures for automata, lexical trees, etc. While it thus has a much broader scope than Edison, we will only consider those parts that are directly comparable.

Baire is still a work in progress and, while it features more implementations than Edison, it does not yet have an as mature framework of interfaces (corresponding to type classes in Haskell). At the time of writing, only a Set and a Map interface exist, where the former corresponds to Edison's `OrdSet` and the latter to `FiniteMap`. Part of this reduction from eight type classes each, as in Edison, to just one each, is due to the fact that Caml provides a total ordering for every data type (except functions), so that it is not necessary to provide interfaces for implementations that do not assume the presence of an ordering. Furthermore, Baire demands observability for all implementations, which again halves the number of interfaces. Baire does not yet feature interfaces for bags or sequences, but these are expected to appear in future releases.

Another library worth mentioning is the *SML/NJ Library*, which includes type signatures for ordered sets and maps which are similar to `Set` and `FiniteMap`, respectively, but require the elements (keys, respectively) to have an ordering relation defined and will keep their contents sorted. The same three implementations are adapted to both signatures, namely an ordered list,

[1]The Functional Graph Library is available for Haskell and Standard ML and focuses on operations dealing with graphs.

a splay tree and size balanced tree as described by Adams [Ada93].

For most other functional programming languages, all data structure implementations are more or less stand-alone and lack a framework tying them together, although they sometimes are bundled in libraries. Examples include *Elib*, the Emacs Lisp library, which includes stack, queue, doubly linked list, binary tree and AVL tree implementations, and *SLIB*, a portable Scheme library, which contains among others a deque (alas called queue) and a priority queue.

Last but not least, we want to mention *DData* by Daan Leijen, a collection of data structures for Haskell including a bag, a set and a map (based on size balanced trees), as well as specialisations of these for integers (based on big-endian patricia trees). These are not yet part of any type framework, but should be easy to adapt to Edison.

4.1.2 Java Collection Framework

The Java Collection Framework is part of the Java 2 platform. As Java does not provide a way of type-parametrisation[2], the collection classes store their elements as `Object`s, the base of Javas class hierarchy. While this allows arbitrary objects to be stored, it also means that retrieved elements have to be type-casted to their original type, an operation that may cause type errors at runtime.

The JCF defines a total of six interfaces, the most basic of which is `Collection`. This is extended to `Set`, `List` and `SortedSet`. Additionally, the interfaces `Map` and `SortedMap` do not extend `Collection`, but stand alone. These interfaces are comparable with the respective Edsion type classes and correspond as depicted in table 4.1. One of the design goals of the JCF has

JCF interface	ordered	unique	Edison type class
List	n.a.	n.a.	Sequence
Collection	-	-	Coll
Set	-	X	Set
SortedSet	X	X	OrdSet
Map	-	X	FiniteMap
SortedMap	X	X	SortedMap

Table 4.1: Correspondence of JCF interfaces and Edison type classes by uniquness and orderedness

been to keep it simple, which is why it consists of a lower number of interfaces than Edison and furthermore, the interfaces contain fewer operations. In particular, all higher-order functions (like `map`, `foldr` etc.) are missing, although they could have been implemented by means of visitor objects that provide the functions to be applied to the elements.

Instead, all collections in the JCF provide iterators that allow traversal over the structure while hiding implementation details. The iterators come in two flavors, one that allows forward traversal and removal of the current element only, and one that allows also backward traversal, adding of new elements and replacing the current element.

[2]This probably will change with the introduction of generics in Java 1.5

4.1.3 Standard Template Library

As the name suggests, the STL makes extensive use of the template facility in C++. While this ensures type safety at compile-time, the template concept has certain problems, as one cannot explicitly restrict the allowed template parameters. Hence, if a class or function template accepts a type as parameter, it can be used with any type. Any assumptions the class/function makes about that type are implicit and may result in confusing error messages if not fulfilled. In the documentation of the STL, the term *concept* is used to categorise types in a way that is very similar to type-classes in Haskell with the important difference that these concepts have no language support.

Contrary to Edison, the container classes in the STL provide only those operations that can be supported efficiently, which gives rise to a multitude of concepts, and we will refrain from a detailed discussion. What is worth noticing however, is that all concepts again demand observability.

Like the JCF, iterators are used in the STL to allow traversal of containers while abstracting away implementation details. Again, iterators only provide those operations they can support efficiently, resulting in a large number of iterator concepts. All operations that go beyond mere adding and removing of single elements, like searching specific elements, applying a function to all elements, partitioning etc., are implemented as function templates that operate on iterators and are therefore independent of the underlying container implementation they are invoked on. This simplifies development of new container classes, as they only need to provide a very limited number of operations and an iterator, and all exiting algorithm can automatically be used.

4.1.4 Noteworthy Differences to Edison

Edison seems to be the only data structure library that allows containers to be non-observable. However, all implementations currently available in Edison are observable. This casts some doubt on the usefulness of the observability concept. In the Edison manual, Okasaki gives two examples of possible implementations that are non-observable: A bag represented as a mapping from elements to counts, and a collection of elements for which a hash function is defined that produces unique values (but is not reversible), where instead of the elements only their hashes are stored. However, as these are not yet implemented, it cannot be estimated whether they are in some respect superior to other, observable data structures, or to what extent non-observable data structures are actually useful in practice. It should be carefully evaluated whether the possibility of non-observable collections adds any real benefits, or just complexity.

Another interesting point is that while iterators have proven useful in practice for imperative/object oriented collection classes to allow traversal of the collection without revealing any internals, their functional counterparts employ different approaches. They provide higher-order functions that perform some specific traversal (e.g. `foldr`, `map`), allow element-wise decomposition (e.g. `lview`, `minView`) and provide a way to convert to a canonical form, namely lists. The problem with this is that the element-wise decomposition usually incurs a lot of overhead, as it might have to restructure the collection to return the remaining elements. And also conversion to a list may result in inefficiency, as the list obviously has to contain the elements in the correct order, while for e.g. `map`, the order is irrelevant as long as it is preserved. This makes it desirable to have a higher-order traversal operation for each form of traversal one might wish for. As has become clear during this project, this causes a lot of work on the side of the implementor of a data structure. However, iterators seem to be rather imperative in nature. Consider for example an iterator that allows, among other things, the deletion of the current element. In an imperative setting, this would update the collection to no longer contain the removed element and the iterator to point to next valid element, and would not have to return anything. In a functional setting, both the new collection and the new iterator would have to be returned. Now suddenly the user has to deal with both the iterator and the collection at the same time, which does not seem beneficial at all. While this might be hidden from the user using monads, it is still a source of inefficiency. This becomes apparent for example considering the simple queue and deque implementations using two lists internally (as in sections 2.2, 2.3, 3.2, 3.3). Here, `map` and all the `fold` operations do not need to `reverse` the rear list, while an iterator most probably would have to.

4.2 Future Work

The most important future work is easily determined: Do the same for the collections and associations parts what has been done in the course of this project for sequences, i.e. provide a detailed analysis of the existing implementations together with a condensed overview and provide new implementations to fill the present gaps. Integration of Daan Leijen's DData package into Edison could do well as starting point. A possible reduction of work can be achieved by taking advantage of the fact that collections and associations are tightly coupled and it is often possible to implement one in terms of the other, e.g. a map could be a set of key-value-pairs or a set could be a key-to-unit map. Although this approach is very tempting, care must be taken not to inflict efficiency penalties with overheads caused by wrapping of arguments and unwrapping of results when calling the underlying implementation.

After suitable data structures have been selected, it should be checked whether any of them exploits non-observability and, if not, the framework might be simplified by removing the non-observable classes. This should, however, only be done after very careful consideration, as to not limit future possibilities.

To make Edison attractive to programmers, finally the manual has to be extended to include the complexity information about the different implementations in a condensed form, like the table in appendix A, and their main characteristics, as to allow easy selection of the data structure most suitable for a certain task. This might also be supplemented with benchmarking results, possibly gathered with the help of the Auburn tool.

4.3 Conclusions

Edison has good chances of becoming the STL for Haskell and rendering an extremely useful library for a broad range of Haskell applications. Despite several gaps, already today, it provides interesting data structures in a framework that makes them easy to use. However, at present Edison is rarely used, mainly due to the lack of documentation of the existing implementations.

With this thesis, we successfully solved this problem for the `Sequence`s part of Edison. Although most of the data structures were based on available theoretical work, their analysis proved extremely labour-intensive, as most papers are limited to a description of about five main operations, while a `Sequence` in Edison provides more than 50. Fortunately, it was often possible to adapt the known arguments to the remaining operations. Noteworthy exceptions include the `drop` operation for the `BinaryRandList`[3] and the analysis of the `JoinList`, for which only much

[3]In the source code, Okasaki added the comment "`drop is O(log^2 n) instead of O(log n)??`", we showed

weaker theoretical works had existed before.

We furthermore implemented four new `Sequence` implementations, rounding off that part of Edison. Of course, we analysed and documented these as carefully as the existing ones. To increase confidence in their correctness, we made use of QuickCheck. In order to do so, we replaced the property definitions already part of Edison with new ones, allowing polymorphic usage and checking of structural invariant preservation as well.

Unfortunately, due to the limited scope of this project, we could not tackle the collections and associative collections parts of Edison, nor do any benchmarking to undermine the theoretical work to be of practical use. We hope however, that we succeded in increasing the usefulness of Edison to a level where it is applied more often, and thus enters the mental scope of more Haskell programmers that might fill in the remiaing gaps.

that it is $\mathcal{O}(\log n)$.

Appendix A

Overview of Time Complexities

The variables for the size of the input used in tables A.2 and A.3 are defined as follows:

n	length of the sequence for functions with only one sequence among their arguments
n_1, n_2, n_3	length of first, second, third sequence for functions with two or three sequences as arguments
n_{min}, n_{max}	minimum/maximum of n_1, n_2, n_3
i, j	first, second integer argument (but at least 1 and at most n)
T_f	time complexity of function argument
m_i	length of the ith sequence in the argument of `concat` *or* length of the result of applying `f` to the ith element for `concatMap f`
m	$\sum_{i=1}^{n} m_i$, i.e. length of the result of `concat` or `concatMap`

Table A.1: definition of the input size variables

All expressions are asymptotic bounds, the $\mathcal{O}()$ has been omitted for sake of brevity.

`empty`, `single`, `null`	$\mathcal{O}(1)$
`lview`	maximum of `lhead` and `ltail`
`rview`	maximum of `rhead` and `rtail`
`map`, `foldr`, `foldl`, `foldr1`, `foldl1`, `reducer`, `reducel`, `reduce1`, `tabulate`, `mapWithIndex`, `foldrWithIndex`, `foldlWithIndex`, `filter`, `partition`, `takeWhile`, `dropWhile`, `splitWhile`, `unzipWith`, `unzipWith3`	$n \cdot T_f$
`adjust`	same as `update` plus T_f
`lookupM`, `lookupWithDefault`	same as `lookup`
`zip3`	same as `zip`
`zipWith`, `zipWith3`	same as `zip` plus $n_{min} \cdot T_f$
`unzip3`	same as `unzip`

Table A.2: Common time complexities of all `Sequence` implementations

Table A.3: Time complexities of `Sequence` operations

	default	`ListSeq`	`SimpleQueue`	`BankersQueue`	`MyersStack`	`RandList`	`BinaryRandList`	`JoinList`	`BraunSeq`	`SimpleDeque`	`BankersDeque`	`CatenableList`	`CatenableDeque`
Usage[a]		p	e	p	p	p	p	e	p	e	p	p	p
`cons`	1	-	-	-	-	-	$\log n$	-	$\log n$	-	-	-	-
`snoc`	n	-	1	1	-	-	-	1	$\log^2 n$	1	1	1	1
`append`	n_1	-	-	-	-	-	$n_1 + \log n_2$	1	$n_1 \log n_2$	-	n_{min}	1	1
`lhead`	1	-	-	-	-	-	$\log n$	n	-	-	-	-	-
`ltail`	1	-	-	-	-	-	$\log n$	-	$\log n$	-	-	-	-
`rhead`	n	-	-	-	$\log n$	$\log n$	$\log n$	-	$\log^2 n$	1	1	-	1
`rtail`	n	-	-	-	-	-	-	-	$\log^2 n$	1	1	-	1
`size`	n	-	-	1	$\log n$	$\log n$	$\log n$	-	$\log^2 n$	-	1	-	-
`concat`	$n + m$	-	-	-	-	-	-	n	$n + m \log m$	-	-	n	n
`reverse`	n	-	-	-	-	-	-	-	-	1	1	-	-
`reverseOnto`	n_1	-	-	-	-	-	$n_1 + \log n_2$	-	$n_1 \log n_2$	-	n_{min}	-	-
`fromList`	n	1	1	-	-	-	-	-	-	-	-	1	-
`toList`	n	-	-	-	-	-	-	-	-	-	-	-	-
`concatMap`	$n \cdot T_f + m$	-	-	-	-	-	-	$n \cdot T_f$	$n \cdot T_f + m \log m$	-	-	$n \cdot T_f$	$n \cdot T_f$
`copy`	1	-	i	-	-	$\log i$	$\log i$	$\log i$	$\log i$	i	-	1	1
`inBounds`	i	-	n	1	$\min(i, \log n)$	$\log i$	$\log i$	n	$\log i$	-	1	-	-
`lookup`	i	-	n	-	$\min(i, \log n)$	$\min(i, \log n)$	$\log n$	n	$\log i$	n	-	-	-
`update`	i	-	n	-	-	$\min(i, \log n)$	$\log n$	-	$\log i$	-	-	-	-
`take`	i	-	-	-	-	-	$i + \log n$	-	i	-	$\min(i, n-i)$	-	-
`drop`	i	-	n	-	$\min(i, \log n)$	$\min(i, \log n)$	$\log n$	-	$i \log n$	-	$\min(i, n-i)$	-	-
`splitAt`	i	-	n	-	-	-	$i + \log n$	-	$i \log n$	-	$\min(i, n-i)$	-	-
`subseq`	$i + j$	-	-	-	$\min(i, \log n)+j$	$\min(i, \log n)+j$	$j + \log n$	-	$j + i \log n$	-	-	-	-
`zip`	n_{min}	-	-	-	-	-	$n_{min} + \log n_{max}$	-	-	-	-	-	-
`unzip`	n	-	-	-	-	-	-	-	-	-	-	-	-

[a] e=ephemeral, p=persistent

Appendix B

Parametricity Properties for `reducer`

Using the framework set out in [Wad89], we can derive a property for functions of the same type as `reducer`, i.e. `Sequence s ⇒ (a → a → a) → a → s a → a`.

$$(\mathrm{red}, \mathrm{red}) \in \forall X \,.\, (X \to X \to X) \to X \to X^* \to X$$

With a function $a : A \to A'$, we can now start the derivation by eliminating the $\forall$ quantifier.

$$(\mathrm{red}_A, \mathrm{red}_{A'}) \in (a \to a \to a) \to a \to a^* \to a$$

We can then apply the rule for $\to$ two times to arrive at

$$\forall (\oplus, \oplus') \in (a \to a \to a) \,.\, \forall (u, u') \in a \,.\, (\mathrm{red}_A\ (\oplus)\ u, \mathrm{red}_{A'}\ (\oplus')\ u') \in a^* \to a.$$

We can rewrite the outer quantification by using

$$\begin{array}{ll} & (\oplus, \oplus') \in (a \to a \to a) \\ \Leftrightarrow & \forall (x, x') \in a \,.\, ((\oplus)x, (\oplus')x') \in (a \to a) \\ \Leftrightarrow & \forall (x, x') \in a \,.\, \forall (y, y') \in a \,.\, (x \oplus y, x \oplus' y') \in a \\ \Leftrightarrow & \forall x, y \in A \,.\, \forall x', y' \in A' \,.\, a\ x = x' \wedge a\ y = y' \Rightarrow a\ (x \oplus y) = x' \oplus' y' \\ \Leftrightarrow & \forall x, y \in A \,.\, a\ (x \oplus y) = (a\ x) \oplus' (a\ y) \end{array}$$

which gives us

$$\begin{array}{l} \forall \oplus \in (A \to A \to A) \,.\, \forall \oplus' \in (A' \to A' \to A') \,. \\ \quad (\forall x, y \in A \,.\, a\ (x \oplus y) = (a\ x) \oplus' (a\ y)) \\ \quad\quad \Rightarrow \forall (u, u') \in a \,.\, (\mathrm{red}_A\ (\oplus)\ u, \mathrm{red}_{A'}\ (\oplus')\ u') \in a^* \to a \end{array}$$

or, equivalently,

$$\begin{array}{l} \forall \oplus \in (A \to A \to A) \,.\, \forall \oplus' \in (A' \to A' \to A') \,. \\ \quad (\forall x, y \in A \,.\, a\ (x \oplus y) = (a\ x) \oplus' (a\ y)) \\ \quad\quad \Rightarrow \forall u \in A \,.\, (\mathrm{red}_A\ (\oplus)\ u, \mathrm{red}_{A'}\ (\oplus')\ (a\ u)) \in a^* \to a. \end{array}$$

Eliminating the last $\to$, we thereby get

$$\begin{array}{l} \forall \oplus \in (A \to A \to A) \,.\, \forall \oplus' \in (A' \to A' \to A') \,. \\ \quad (\forall x, y \in A \,.\, a\ (x \oplus y) = (a\ x) \oplus' (a\ y)) \\ \quad\quad \Rightarrow \forall u \in A \,.\, \forall (xs, xs') \in a^* \,.\, (\mathrm{red}_A\ (\oplus)\ u\ xs, \mathrm{red}_{A'}\ (\oplus')\ (a\ u)\ xs') \in a^* \to a \end{array}$$

which can be rewritten to the more convenient form

$$\begin{array}{l}\forall\oplus \in (A \to A \to A) \,.\, \forall\oplus' \in (A' \to A' \to A')\,. \\ \quad (\forall x, y \in A \,.\, a\ (x \oplus y) = (a\ x) \oplus' (a\ y)) \\ \qquad \Rightarrow \forall u \in A \,.\, \forall xs \in A^* \,.\, a\ (\mathrm{red}_A\ (\oplus)\ u\ xs) = \mathrm{red}_{A'}\ (\oplus')\ (a\ u)\ (a\ xs)).\end{array}$$

Appendix C

Complete Source Code of Module `CheckSeq`

```
module CheckSeq where

import Prelude hiding (concat,reverse,map,concatMap,foldr,foldl,foldr1,foldl1,
                       filter,takeWhile,dropWhile,lookup,take,drop,splitAt,
                       zip,zip3,zipWith,zipWith3,unzip,unzip3,null)
import qualified Prelude

import EdisonPrelude(Maybe2(Just2,Nothing2))
import QuickCheck

import Sequence

instance Show (a → b) where
  show f = "<function>"

class Sequence s ⇒ CheckableSequence s where
  invariant :: s a → Bool
  arbitraryWith :: [a] → Gen (s a)

prop_lview :: (CheckableSequence s, Eq a) ⇒ s a → Bool
prop_lview xs = case lview xs of
                  Nothing2 → True
                  Just2 _ xs' → invariant xs'

prop_empty :: (Sequence s, Eq a, Eq (s a)) ⇒ (s a → Bool) → Bool
prop_empty inv = inv xs && case lview xs of
                             Nothing2 → True
                             Just2 _ _ → False
  where xs = empty
```

```
equalsDef :: (Sequence s, Eq a) ⇒ s a → s a → Bool
equalsDef xs ys = case lview xs of
                    Nothing2 → case lview ys of
                                  Nothing2 → True
                                  Just2 _ _ → False
                    Just2 x xs' → case lview ys of
                                     Nothing2 → False
                                     Just2 y ys' → x==y && equalsDef xs' ys'

prop_equals :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → s a → Bool
prop_equals xs ys = (xs==ys) == equalsDef xs ys

prop_cons :: (CheckableSequence s, Eq a, Eq (s a)) ⇒ s a → a → Bool
prop_cons xs x = lview xxs == Just2 x xs && invariant xxs
  where xxs = cons x xs

nullDef :: Sequence s ⇒ s a → Bool
nullDef xs = case lview xs of
               Nothing2 → True
               Just2 _ _ → False

prop_null :: (Sequence s) ⇒ s a → Bool
prop_null xs = null xs == nullDef xs

singleDef :: Sequence s ⇒ a → s a
singleDef x = cons x empty

prop_single :: (Sequence s, Eq a, Eq (s a)) ⇒ (s a → Bool) → a → Bool
prop_single inv x = inv xs && xs == singleDef x
  where xs = single x

snocDef :: Sequence s ⇒ s a → a → s a
snocDef xs y = case lview xs of
                 Nothing2 → single y
                 Just2 x xs' → cons x (snocDef xs' y)

prop_snoc :: (CheckableSequence s, Eq a, Eq (s a)) ⇒ s a → a → Bool
prop_snoc xs x = invariant xs' && xs' == snocDef xs x
  where xs' = snoc xs x

appendDef :: Sequence s ⇒ s a → s a → s a
appendDef xs ys = case lview xs of
                    Nothing2 → ys
                    Just2 x xs' → cons x (append xs' ys)

prop_append :: (CheckableSequence s, Eq a, Eq (s a))
               ⇒ s a → s a → Bool
prop_append xs ys = invariant xsys && xsys == appendDef xs ys
  where xsys = append xs ys
```

```
fromListDef :: Sequence s ⇒ [a] → s a
fromListDef [] = empty
fromListDef (x:xs) = cons x (fromListDef xs)

prop_fromList :: (Sequence s, Eq a, Eq (s a)) ⇒ (s a → Bool) → [a] → Bool
prop_fromList inv l = inv s && s == fromListDef l
  where s = fromList l

copyDef :: Sequence s ⇒ Int → a → s a
copyDef n x | n ≤ 0 = empty
            | otherwise = cons x (copyDef (n-1) x)

prop_copy :: (Sequence s, Eq a, Eq (s a)) ⇒ (s a → Bool) → Int → a → Bool
prop_copy inv n x = inv xs && xs == copyDef n x
  where xs = copy n x

tabulateDef :: Sequence s ⇒ Int → (Int → a) → s a
tabulateDef n f | n ≤ 0 = empty
                | otherwise = snoc (tabulateDef (n-1) f) (f (n-1))

prop_tabulate :: (Sequence s, Eq a, Eq (s a)) ⇒ (s a → Bool) → Int
                                                → (Int → a) → Bool
prop_tabulate inv n f = inv s && s == tabulateDef n f
    where s = tabulate n f

lheadDef :: Sequence s ⇒ s a → a
lheadDef xs = let Just2 x _ = lview xs in x

prop_lhead :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → Property
prop_lhead xs = not (null xs) ⟹ lhead xs == lheadDef xs

ltailDef :: Sequence s ⇒ s a → s a
ltailDef xs = case lview xs of
                Nothing2 → empty
                Just2 _ xs' → xs'

prop_ltail :: (CheckableSequence s, Eq a, Eq (s a)) ⇒ s a → Bool
prop_ltail xs = invariant s && s == ltailDef xs
  where s = ltail xs

rviewDef :: Sequence s ⇒ s a → Maybe2 (s a) a
rviewDef xs = case lview xs of
                Nothing2 → Nothing2
                Just2 x' xs' → case rviewDef xs' of
                                 Nothing2 → Just2 empty x'
                                 Just2 xs'' x'' → Just2 (cons x' xs'') x''

prop_rview :: (CheckableSequence s, Eq a, Eq (s a)) ⇒ s a → Bool
prop_rview xs =
    case xsx of
        Nothing2    → rviewDef xs == xsx
        Just2 xs' x → invariant xs' && rviewDef xs == xsx
    where xsx = rview xs

rheadDef :: Sequence s ⇒ s a → a
rheadDef xs = let Just2 _ x = rview xs in x

prop_rhead :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → Property
prop_rhead xs = not (null xs) ⟹ rhead xs == rheadDef xs

rtailDef :: Sequence s ⇒ s a → s a
rtailDef xs = case rview xs of
                Nothing2 → empty
                Just2 xs' _ → xs'

prop_rtail :: (CheckableSequence s, Eq a, Eq (s a)) ⇒ s a → Bool
prop_rtail xs = invariant s && s == rtailDef xs
  where s = rtail xs

sizeDef :: Sequence s ⇒ s a → Int
sizeDef xs = case lview xs of
               Nothing2 → 0
               Just2 _ xs' → 1 + sizeDef xs'

prop_size  :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → Property
prop_size xs = collect (size xs) $ size xs == sizeDef xs

toListDef :: Sequence s ⇒ s a → [a]
toListDef xs = case lview xs of
                 Nothing2 → []
                 Just2 x xs' → x : toListDef xs'

prop_toList :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → Bool
prop_toList xs = toList xs == toListDef xs

concatDef :: Sequence s ⇒ s (s a) → s a
concatDef xss = case lview xss of
                  Nothing2 → empty
                  Just2 xs xss' → append xs (concatDef xss')

prop_concat :: (Sequence s, Eq a, Eq (s a)) ⇒ (s a → Bool) → s (s a)
                                                    → Bool
prop_concat inv xss = inv res && res == concatDef xss
  where res = concat xss

reverseDef :: Sequence s ⇒ s a → s a
reverseDef xs = case lview xs of
```

```
                  Nothing2 → empty
                  Just2 x xs' → snoc (reverseDef xs') x

prop_reverse :: (CheckableSequence s, Eq a, Eq (s a)) ⇒ s a → Bool
prop_reverse xs = invariant rev && rev == reverseDef xs
  where rev = reverse xs

reverseOntoDef :: Sequence s ⇒ s a → s a → s a
reverseOntoDef xs ys = append (reverse xs) ys

prop_reverseOnto  :: (CheckableSequence s, Eq a, Eq (s a))
                     ⇒ s a → s a → Bool
prop_reverseOnto xs ys = invariant rev && rev == reverseOntoDef xs ys
  where rev = reverseOnto xs ys

mapDef :: Sequence s ⇒ (a → b) → s a → s b
mapDef f xs = case lview xs of
                Nothing2 → empty
                Just2 x xs' → cons (f x) (mapDef f xs')

prop_map :: (CheckableSequence s, Eq a, Eq (s a)) ⇒ s a → (a → a) → Bool
prop_map xs f = invariant xs' && xs' == mapDef f xs
  where xs' = map f xs

concatMapDef :: Sequence s ⇒ (a → s b) → s a → s b
concatMapDef f xs = concat (map f xs)

prop_concatMap :: (CheckableSequence s, Eq a, Eq (s a))
                 ⇒ s a → (a → s a) → Bool
prop_concatMap xs f = invariant xs' && xs' == concatMapDef f xs
  where xs' = concatMap f xs

foldrDef :: Sequence s ⇒ (a → b → b) → b → s a → b
foldrDef f e xs = case lview xs of
                    Nothing2 → e
                    Just2 x xs' → f x (foldrDef f e xs')

prop_foldr :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → (a → a → a) → a
                                             → Bool
prop_foldr xs f e = foldr f e xs == foldrDef f e xs

foldlDef :: Sequence s ⇒ (b → a → b) → b → s a → b
foldlDef f e xs = case lview xs of
                    Nothing2 → e
                    Just2 x xs' → foldlDef f (f e x) xs'

prop_foldl :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → (a → a → a) → a
                                             → Bool
prop_foldl xs f e = foldl f e xs == foldlDef f e xs
```

```
foldr1Def :: Sequence s ⇒ (a → a → a) → s a → a
foldr1Def f xs = let Just2 xs' x = rview xs in foldr f x xs'

prop_foldr1 :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → (a → a → a)
                                               → Property
prop_foldr1 xs f = not (null xs) ⟹ foldr1 f xs == foldr1Def f xs

foldl1Def :: Sequence s ⇒ (a → a → a) → s a → a
foldl1Def f xs = let Just2 x xs' = lview xs in foldl f x xs'

prop_foldl1 :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → (a → a → a)
                                               → Property
prop_foldl1 xs f = not (null xs) ⟹ foldl1 f xs == foldl1Def f xs

reducerDef :: Sequence s ⇒ (a → a → a) → a → s a → a
reducerDef = foldr

prop_reducer :: (Sequence s, Eq a, Eq (s a), Arbitrary a)
                ⇒ s a → a → Bool
prop_reducer xs' e' = let xs = map (λx' → [x']) xs'
                          e = [e']
                      in reducer (++) e xs == reducerDef (++) e xs

reducelDef :: Sequence s ⇒ (a → a → a) → a → s a → a
reducelDef = foldl

prop_reducel :: (Sequence s, Eq a, Eq (s a), Arbitrary a)
                ⇒ s a → a → Bool
prop_reducel xs' e' = let xs = map (λx' → [x']) xs'
                          e = [e']
                      in reducel (++) e xs == reducelDef (++) e xs

reduce1Def :: Sequence s ⇒ (a → a → a) → s a → a
reduce1Def = foldr1

prop_reduce1 :: (Sequence s, Eq a, Eq (s a), Arbitrary a) ⇒ s a → Property
prop_reduce1 xs' = let xs = map (λx' → [x']) xs'
                   in not (null xs') ⟹ reduce1 (++) xs == reduce1Def (++) xs

takeDef :: Sequence s ⇒ Int → s a → s a
takeDef i xs | i ≤ 0 = empty
             | i ≥ size xs = xs
             | otherwise = let Just2 x xs' = lview xs
                           in cons x (takeDef (i-1) xs')

prop_take :: (CheckableSequence s, Eq a, Eq (s a)) ⇒ s a → Int → Bool
prop_take xs i = invariant xs' && xs' == takeDef i xs
  where xs' = take i xs
```

```
dropDef :: Sequence s ⇒ Int → s a → s a
dropDef i xs | i ≤ 0 = xs
             | i ≥ size xs = empty
             | otherwise = dropDef (i-1) (ltail xs)

prop_drop :: (CheckableSequence s, Eq a, Eq (s a)) ⇒ s a → Int → Bool
prop_drop xs i = invariant xs' && xs' == dropDef i xs
  where xs' = drop i xs

splitAtDef :: Sequence s ⇒ Int → s a → (s a, s a)
splitAtDef i xs = (take i xs, drop i xs)

prop_splitAt :: (CheckableSequence s, Eq a, Eq (s a)) ⇒ s a → Int → Bool
prop_splitAt xs i = invariant xs' && invariant xs''
                    && (xs', xs'') == splitAtDef i xs
  where (xs', xs'') = splitAt i xs

subseqDef :: Sequence s ⇒ Int → Int → s a → s a
subseqDef i j xs = take j (drop i xs)

prop_subseq :: (CheckableSequence s, Eq a, Eq (s a))
               ⇒ s a → Int → Int → Bool
prop_subseq xs i j = invariant xs' && xs' == subseqDef i j xs
  where xs' = subseq i j xs

filterDef :: Sequence s ⇒ (a → Bool) → s a → s a
filterDef p xs = case lview xs of
                   Nothing2 → empty
                   Just2 x xs' → if p x then cons x (filterDef p xs')
                                        else filterDef p xs'

prop_filter :: (CheckableSequence s, Eq a, Eq (s a))
               ⇒ s a → (a → Bool) → Bool
prop_filter xs p = invariant xs' && xs' == filterDef p xs
  where xs' = filter p xs

partitionDef :: Sequence s ⇒ (a → Bool) → s a → (s a, s a)
partitionDef p xs = (filter p xs, filter (not . p) xs)

prop_partition :: (CheckableSequence s, Eq a, Eq (s a))
                  ⇒ s a → (a → Bool) → Bool
prop_partition xs p = invariant xs' && invariant xs''
                      && (xs', xs'') == partitionDef p xs
  where (xs', xs'') = partition p xs

takeWhileDef :: Sequence s ⇒ (a → Bool) → s a → s a
takeWhileDef p xs = case lview xs of
                      Nothing2 → empty
                      Just2 x xs' → if p x then cons x (takeWhileDef p xs')
                                              else empty

prop_takeWhile :: (CheckableSequence s, Eq a, Eq (s a))
                  ⇒ s a → (a → Bool) → Bool
prop_takeWhile xs p = invariant xs' && xs' == takeWhileDef p xs
  where xs' = takeWhile p xs

dropWhileDef :: Sequence s ⇒ (a → Bool) → s a → s a
dropWhileDef p xs = case lview xs of
                      Nothing2 → empty
                      Just2 x xs' → if p x then (dropWhileDef p xs')
                                              else xs

prop_dropWhile :: (CheckableSequence s, Eq a, Eq (s a))
                  ⇒ s a → (a → Bool) → Bool
prop_dropWhile xs p = invariant xs' && xs' == dropWhileDef p xs
  where xs' = dropWhile p xs

splitWhileDef :: Sequence s ⇒ (a → Bool) → s a → (s a, s a)
splitWhileDef p xs = (takeWhile p xs, dropWhile p xs)

prop_splitWhile :: (CheckableSequence s, Eq a, Eq (s a))
                   ⇒ s a → (a → Bool) → Bool
prop_splitWhile xs p = invariant xs' && invariant xs''
                       && (xs', xs'') == splitWhileDef p xs
  where (xs', xs'') = splitWhile p xs

inBoundsDef :: Sequence s ⇒ s a → Int → Bool
inBoundsDef xs i = 0≤i && i<size xs

prop_inBounds :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → Int → Bool
prop_inBounds xs i = inBounds xs i == inBoundsDef xs i

lookupDef :: Sequence s ⇒ s a → Int → a
lookupDef xs 0 = lhead xs
lookupDef xs i = lookupDef (ltail xs) (i-1)

prop_lookup :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → Int → Property
prop_lookup xs i = inBounds xs i ⟹ lookup xs i == lookupDef xs i

lookupMDef :: Sequence s ⇒ s a → Int → Maybe a
lookupMDef xs i | not (inBounds xs i) = Nothing
lookupMDef xs 0 = Just (lhead xs)
lookupMDef xs i = lookupMDef (ltail xs) (i-1)

prop_lookupM :: (Sequence s, Eq a, Eq (s a)) ⇒ s a → Int → Bool
prop_lookupM xs i = lookupM xs i == lookupMDef xs i
```

```
lookupWithDefaultDef :: Sequence s ⇒ a → s a → Int → a
lookupWithDefaultDef d xs i | not (inBounds xs i) = d
lookupWithDefaultDef d xs 0 = lhead xs
lookupWithDefaultDef d xs i = lookupWithDefaultDef d (ltail xs) (i-1)

prop_lookupWithDefault :: (Sequence s, Eq a, Eq (s a))
                          ⇒ s a → a → Int → Bool
prop_lookupWithDefault xs d i
  = lookupWithDefault d xs i == lookupWithDefaultDef d xs i

updateDef :: Sequence s ⇒ Int → a → s a → s a
updateDef i e xs | not (inBounds xs i) = xs
updateDef 0 e xs = cons e (ltail xs)
updateDef i e xs = let Just2 x xs' = lview xs
                   in cons x (updateDef (i-1) e xs')

prop_update :: (CheckableSequence s, Eq a, Eq (s a))
               ⇒ s a → Int → a → Bool
prop_update xs i e = invariant xs' && xs' == updateDef i e xs
  where xs' = update i e xs

adjustDef :: Sequence s ⇒ (a → a) → Int → s a → s a
adjustDef f i xs | not (inBounds xs i) = xs
adjustDef f 0 xs = let Just2 x xs' = lview xs
                   in cons (f x) xs'
adjustDef f i xs = let Just2 x xs' = lview xs
                   in cons x (adjustDef f (i-1) xs')

prop_adjust :: (CheckableSequence s, Eq a, Eq (s a))
               ⇒ s a → (a → a) → Int → Bool
prop_adjust xs f i = invariant xs' && xs' == adjustDef f i xs
  where xs' = adjust f i xs

mapWithIndexDef :: Sequence s ⇒ (Int → a → b) → s a → s b
mapWithIndexDef f xs
  = case rview xs of
      Nothing2 → empty
      Just2 xs' x → snoc (mapWithIndexDef f xs') (f (size xs') x)

prop_mapWithIndex :: (CheckableSequence s, Eq a, Eq (s a))
                     ⇒ s a → (Int → a → a) → Bool
prop_mapWithIndex xs f = invariant xs' && xs' == mapWithIndexDef f xs
  where xs' = mapWithIndex f xs

foldrWithIndexDef :: Sequence s ⇒ (Int → a → b → b) → b → s a → b
foldrWithIndexDef f e xs
  = case rview xs of
      Nothing2 → e
      Just2 xs' x → foldrWithIndexDef f (f (size xs') x e) xs'
```

```
prop_foldrWithIndex :: (Sequence s, Eq a, Eq (s a))
             ⇒ s a → (Int → a → a → a) → a → Bool
prop_foldrWithIndex xs f e = foldrWithIndex f e xs == foldrWithIndexDef f e xs

foldlWithIndexDef :: Sequence s ⇒ (b → Int → a → b) → b → s a → b
foldlWithIndexDef f e xs
  = case rview xs of
      Nothing2 → e
      Just2 xs' x → f (foldlWithIndexDef f e xs') (size xs') x

prop_foldlWithIndex :: (Sequence s, Eq a, Eq (s a))
             ⇒ s a → (a → Int → a → a) → a → Bool
prop_foldlWithIndex xs f e = foldlWithIndex f e xs == foldlWithIndexDef f e xs

zipWithDef :: Sequence s ⇒ (a → b → c) → s a → s b → s c
zipWithDef f xs ys
  | null xs || null ys = empty
  |otherwise = let Just2 x xs' = lview xs
                   Just2 y ys' = lview ys
               in cons (f x y) (zipWithDef f xs' ys')

prop_zipWith :: (CheckableSequence s, Eq a, Eq (s a))
               ⇒ s a → s a → (a → a → a) → Bool
prop_zipWith xs ys f = invariant xys && xys == zipWithDef f xs ys
  where xys = zipWith f xs ys

zipWith3Def :: Sequence s ⇒ (a → b → c → d) → s a → s b → s c → s d
zipWith3Def f xs ys zs
  | null xs || null ys || null zs = empty
  |otherwise = let Just2 x xs' = lview xs
                   Just2 y ys' = lview ys
                   Just2 z zs' = lview zs
               in cons (f x y z) (zipWith3Def f xs' ys' zs')

prop_zipWith3 :: (CheckableSequence s, Eq a, Eq (s a))
                ⇒ s a → s a → s a → (a → a → a → a) → Bool
prop_zipWith3 xs ys zs f = invariant xyzs && xyzs == zipWith3Def f xs ys zs
  where xyzs = zipWith3 f xs ys zs

zipDef :: Sequence s ⇒ s a → s b → s (a, b)
zipDef xs ys = zipWith (λx y → (x, y)) xs ys

prop_zip :: (CheckableSequence s, Eq a, Eq (s (a, a))) ⇒ s a → s a → Bool
prop_zip xs ys = invariant xys && xys == zipDef xs ys
  where xys = zip xs ys

zip3Def :: Sequence s ⇒ s a → s b → s c → s (a, b, c)
zip3Def xs ys zs = zipWith3 (λx y z → (x, y, z)) xs ys zs
```

```
prop_zip3 :: (CheckableSequence s, Eq a, Eq (s (a, a, a)))
            ⇒ s a → s a → s a → Bool
prop_zip3 xs ys zs = invariant xyzs && xyzs == zip3Def xs ys zs
  where xyzs = zip3 xs ys zs

unzipDef :: Sequence s ⇒ s (a, b) → (s a, s b)
unzipDef xs = (map fst xs, map snd xs)

prop_unzip :: (Sequence s, Eq a, Eq (s a))
             ⇒ (s a → Bool) → s (a, a) → Bool
prop_unzip inv xys = inv xs && inv ys && (xs, ys) == unzipDef xys
  where (xs, ys) = unzip xys

unzip3Def :: Sequence s ⇒ s (a, b, c) → (s a, s b, s c)
unzip3Def xs = (map fst3 xs, map snd3 xs, map thd3 xs)
  where fst3 (x, _, _) = x
        snd3 (_, y, _) = y
        thd3 (_, _, z) = z

prop_unzip3 :: (Sequence s, Eq a, Eq (s a))
             ⇒ (s a → Bool) → s (a, a, a) → Bool
prop_unzip3 inv xyzs = inv xs && inv ys && inv zs
                       && (xs, ys, zs) == unzip3Def xyzs
  where (xs, ys, zs) = unzip3 xyzs

unzipWithDef :: Sequence s ⇒ (a → b) → (a → c) → s a → (s b, s c)
unzipWithDef f g xs = (map f xs, map g xs)

prop_unzipWith :: (CheckableSequence s, Eq a, Eq (s a))
                 ⇒ s a → (a → a) → (a → a) → Bool
prop_unzipWith xs f g = invariant xs' && invariant xs''
                        && (xs', xs'') == unzipWithDef f g xs
  where (xs', xs'') = unzipWith f g xs

unzipWith3Def :: Sequence s
                ⇒ (a → b) → (a → c) → (a → d) → s a → (s b, s c, s d)
unzipWith3Def f g h xs = (map f xs, map g xs, map h xs)

prop_unzipWith3 :: (CheckableSequence s, Eq a, Eq (s a))
                  ⇒ s a → (a → a) → (a → a) → (a → a) → Bool
prop_unzipWith3 xs f g h = invariant xs' && invariant xs'' && invariant xs'''
                           && (xs', xs'', xs''') == unzipWith3Def f g h xs
  where (xs', xs'', xs''') = unzipWith3 f g h xs

checkseq :: (Sequence s, Eq a, Arbitrary (s (s a)), Show (s (s a)),
             Arbitrary a, Show a, Eq (s a), Arbitrary (s a), Show (s a),
             Arbitrary (s (a, a)), Show (s (a, a)), Eq (s (a, a)),
             Arbitrary (s (a, a, a)), Eq (s (a, a, a)), CheckableSequence s,
             Show (s (a, a, a)))
            ⇒ (s a → Bool) → IO ()

checkseq inv = do putStr "Checking␣arbitrary...␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs inv)
                  putStr "Checking␣arbitraryWith...␣␣␣␣␣"
                  quickCheck (λxs → forAllWith xs
                              (λs → property $ inv s && xs == toListDef s))

                  putStr "Checking␣lview␣...␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_lview)
                  putStr "Checking␣empty...␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (prop_empty inv)
                  putStr "Checking␣==␣...␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_equals)

                  putStr "Checking␣==␣(equal␣cases)...␣␣"
                  quickCheck (λxs → forAllWith xs
                              (λs1 → forAllWith xs
                               (λs2 → property $ prop_equals s1 s2)))

                  putStr "Checking␣cons...␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_cons)
                  putStr "Checking␣null...␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_null)
                  putStr "Checking␣single...␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (prop_single inv)
                  putStr "Checking␣snoc...␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_snoc)
                  putStr "Checking␣append...␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_append)
                  putStr "Checking␣fromList...␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (prop_fromList inv)
                  putStr "Checking␣copy...␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (prop_copy inv)
                  putStr "Checking␣tabulate...␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (prop_tabulate inv)
                  putStr "Checking␣lhead...␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_lhead)
                  putStr "Checking␣ltail...␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_ltail)
                  putStr "Checking␣rview...␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_rview)
                  putStr "Checking␣rhead...␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_rhead)
                  putStr "Checking␣rtail...␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_rtail)
                  putStr "Checking␣size...␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
                  quickCheck (forAll seqs prop_size)
```

```
putStr "Checking␣toList...␣␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_toList)
putStr "Checking␣concat...␣␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll gen20 (prop_concat inv))
putStr "Checking␣reverse...␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_reverse)
putStr "Checking␣reverseOnto...␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_reverseOnto)
putStr "Checking␣map...␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_map)

putStr "Checking␣concatMap...␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs
            (λxs → forAll gen20
             (λf → (prop_concatMap xs f))))

putStr "Checking␣foldr...␣␣␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_foldr)
putStr "Checking␣foldl...␣␣␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_foldl)
putStr "Checking␣foldr1...␣␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_foldr1)
putStr "Checking␣foldl1...␣␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_foldl1)
putStr "Checking␣reducer...␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_reducer)
putStr "Checking␣reducel...␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_reducel)
putStr "Checking␣reduce1...␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_reduce1)
putStr "Checking␣take...␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_take)
putStr "Checking␣drop...␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_drop)
putStr "Checking␣splitAt...␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_splitAt)
putStr "Checking␣subseq...␣␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_subseq)
putStr "Checking␣filter...␣␣␣␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_filter)
putStr "Checking␣partition...␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_partition)
putStr "Checking␣takeWhile...␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_takeWhile)
putStr "Checking␣dropWhile...␣␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_dropWhile)
putStr "Checking␣splitWhile...␣␣␣␣␣␣␣␣"
quickCheck (forAll seqs prop_splitWhile)
putStr "Checking␣inBounds...␣␣␣␣␣␣␣␣␣␣"
```

```
   quickCheck (forAll seqs prop_inBounds)
   putStr "Checking␣lookup...␣␣␣␣␣␣␣␣␣␣␣␣"
   quickCheck (forAll seqs prop_lookup)
   putStr "Checking␣lookupM...␣␣␣␣␣␣␣␣␣␣␣"
   quickCheck (forAll seqs prop_lookupM)
   putStr "Checking␣lookupWithDefault...␣"
   quickCheck (forAll seqs prop_lookupWithDefault)
   putStr "Checking␣update...␣␣␣␣␣␣␣␣␣␣␣␣"
   quickCheck (forAll seqs prop_update)
   putStr "Checking␣adjust...␣␣␣␣␣␣␣␣␣␣␣␣"
   quickCheck (forAll seqs prop_adjust)
   putStr "Checking␣mapWithIndex...␣␣␣␣␣␣"
   quickCheck (forAll seqs prop_mapWithIndex)
   putStr "Checking␣foldrWithIndex...␣␣␣␣"
   quickCheck (forAll seqs prop_foldrWithIndex)
   putStr "Checking␣foldlWithIndex...␣␣␣␣"
   quickCheck (forAll seqs prop_foldlWithIndex)
   putStr "Checking␣zipWith...␣␣␣␣␣␣␣␣␣␣␣"
   quickCheck (forAll seqs prop_zipWith)
   putStr "Checking␣zipWith3...␣␣␣␣␣␣␣␣␣␣"
   quickCheck (forAll seqs prop_zipWith3)
   putStr "Checking␣zip...␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
   quickCheck (forAll seqs prop_zip)
   putStr "Checking␣zip3...␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
   quickCheck (forAll seqs prop_zip3)
   putStr "Checking␣unzip...␣␣␣␣␣␣␣␣␣␣␣␣␣"
   quickCheck (prop_unzip inv)
   putStr "Checking␣unzip3...␣␣␣␣␣␣␣␣␣␣␣␣"
   quickCheck (prop_unzip3 inv)
   putStr "Checking␣unzipWith...␣␣␣␣␣␣␣␣␣"
   quickCheck (forAll seqs prop_unzipWith)
   putStr "Checking␣unzipWith3...␣␣␣␣␣␣␣␣"
   quickCheck (forAll seqs prop_unzipWith3)

where seqs = arbitrary
      genWith = λxs → arbitraryWith xs
      forAllWith = λxs prop → forAll (genWith xs) prop
      gen20 :: Arbitrary a ⇒ Gen a
      gen20 = sized (λn → resize (min 20 n) arbitrary)
```

Bibliography

[Ada93] Stephen Adams. Functional Pearls: Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 268–279, N.Y., September 18–21 2000. ACM Press.

[Fre97] Michael L. Fredman. Pairing heaps are suboptimal. Technical Report 97-52, DIMACS, 8 September 1997.

[FSST86] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[HM81] Robert Hood and Robert Melville. Real-time queue operations in Pure LISP. *Information Processing Letters*, 13(2):50–54, November 1981.

[Hoo92a] R. R. Hoogerwoord. A logarithmetic implementation of flexible arrays. In *Mathematics of Program Construction; Second International Conference; Proceedings*, pages 191–207, 1992.

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

[KT95] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 93–102, 29 May–1 June 1995.

[Mor68] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Jrnl. A.C.M.*, 15(4):514–534, October 1968.

[MR98] Graeme E. Moss and Colin Runciman. Auburn: A kit for benchmarking functional data structures. *Lecture Notes in Computer Science*, 1467:141–160, 1998.

[MR99] Graeme E. Moss and Colin Runciman. Automated benchmarking of functional data structures. *Lecture Notes in Computer Science*, 1551:1–15, 1999.

[Mye83] Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–248, December 1983.

[OG98] Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, September 1998.

[Oka95a] Chris Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *36th Annual Symposium on Foundations of Computer Science*, pages 646–654, Milwaukee, Wisconsin, 23–25 October 1995. IEEE.

[Oka95b] Chris Okasaki. Purely functional random-access lists. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 86–95. ACM Press, 1995.

[Oka95c] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[Oka97a] Chris Okasaki. Catenable double-ended queues. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 66–74. ACM Press, 1997.

[Oka97b] Chris Okasaki. Three algorithms on Braun trees. *Journal of Functional Programming*, 7(6):661–666, November 1997. Functional Pearl.

[Oka98] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[Oka99] Chris Okasaki. *Edison User's Guide (Haskell version)*, 1999.

[San90] D. Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of the Third European Symposium on Programming*, number 432 in LNCS, pages 361–376. Springer-Verlag, May 1990.

[ST86] Daniel Dominic Sleator and Robert Endre Tarjan. Self-Adjusting Heaps. *SIAM Journal on Computing*, 15(1):52–69, February 1986.

[Wad88] P. Wadler. Strictness analysis aids time analysis. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principlesof programming languages*, pages 119–132. ACM Press, 1988.

[Wad89] P. Wadler. Theorems for free! In *FPCA'89, London, England*, pages 347–359. ACM Press, September 1989.