Data Structures I

Advanced Functional Programming

Andres Löh (andres@cs.uu.nl)

Universiteit Utrecht

19 May 2005

Overview

Overview

Introduction (Lists)

Arrays

Unboxed types

Queues and deques

Summary and next lecture

Question

What is the most frequently used data structure in Haskell?

Clearly, lists …

Question

What is the most frequently used data structure in Haskell?

Clearly, lists ...

What are lists good for?

$$
\begin{aligned}
&\mathit{head} &&:: [a] \rightarrow a \\
&\mathit{tail} &&:: [a] \rightarrow [a] \\
&(:) &&:: a \rightarrow [a] \rightarrow [a]
\end{aligned}
$$

- These are efficient operations on lists.
- These are the stack operations.

What are lists good for?

$$\begin{aligned}
\mathit{head} &:: [a] \rightarrow a \\
\mathit{tail} &:: [a] \rightarrow [a] \\
(:) &:: a \rightarrow [a] \rightarrow [a]
\end{aligned}$$

- These are efficient operations on lists.
- These are the stack operations.

What are lists good for?

$$
\begin{aligned}
&\mathit{head} &&:: [a] \to a && \text{-- } O(1)\\
&\mathit{tail} &&:: [a] \to [a] && \text{-- } O(1)\\
&(:) &&:: a \to [a] \to [a] && \text{-- } O(1)
\end{aligned}
$$

- These are efficient operations on lists.
- These are the stack operations.

What are lists good for?

$$\begin{aligned}
&\mathit{top} \;::\; [a] \rightarrow a && \text{-- } O(1)\\
&\mathit{pop} \;::\; [a] \rightarrow [a] && \text{-- } O(1)\\
&\mathit{push} \;::\; a \rightarrow [a] \rightarrow [a] && \text{-- } O(1)
\end{aligned}$$

- These are efficient operations on lists.
- These are the stack operations.

Haskell stacks are persistent

A data structure is called persistent if after an operation both the original and the resutling version of the data structure are available.

If not persistent, a data structure is called ephemeral.

- Functional data structures are naturally persistent.
- Imperative data structures are usually ephemeral.
- Persistent data structures are often, but not always, less efficient than ephemeral data structures.

Haskell stacks are persistent

A data structure is called persistent if after an operation both the original and the resutling version of the data structure are available.

If not persistent, a data structure is called ephemeral.

- Functional data structures are naturally persistent.
- Imperative data structures are usually ephemeral.
- Persistent data structures are often, but not always, less efficient than ephemeral data structures.

Haskell stacks are persistent

A data structure is called persistent if after an operation both the original and the resutling version of the data structure are available.

If not persistent, a data structure is called ephemeral.

- Functional data structures are naturally persistent.
- Imperative data structures are usually ephemeral.
- Persistent data structures are often, but not always, less efficient than ephemeral data structures.

Other operations on lists

$$
\begin{array}{lll}
\mathit{snoc} & :: [a] \to a \to [a] & \text{-- } O(n) \\
\mathit{snoc} & = \lambda xs\ x \to xs +\!\!+ [x] & \\
(+\!\!+) & :: [a] \to [a] \to [a] & \text{-- } O(n) \\
\mathit{reverse} & :: [a] \to [a] & \text{-- } O(n)\text{, naively: } O(n^2) \\
\mathit{union} & :: \mathsf{Eq}\ a \Rightarrow [a] \to [a] \to [a] & \text{-- } O(mn) \\
\mathit{elem} & :: \mathsf{Eq}\ a \Rightarrow a \to [a] \to \mathsf{Bool} & \text{-- } O(n)
\end{array}
$$

Although not efficient for these purposes, Haskell lists are frequently used as

- arrays
- queues, deques, catenable queues
- sets
- lookup tables, association lists, finite maps
- …

Why?

Other operations on lists

$$
\begin{array}{lll}
\mathit{snoc} & :: [a] \to a \to [a] & \text{-- } O(n) \\
\mathit{snoc} & = \lambda xs\ x \to xs +\!\!+ [x] & \\
(+\!\!+) & :: [a] \to [a] \to [a] & \text{-- } O(n) \\
\mathit{reverse} & :: [a] \to [a] & \text{-- } O(n)\text{, naively: } O(n^2) \\
\mathit{union} & :: \mathsf{Eq}\ a \Rightarrow [a] \to [a] \to [a] & \text{-- } O(mn) \\
\mathit{elem} & :: \mathsf{Eq}\ a \Rightarrow a \to [a] \to \mathsf{Bool} & \text{-- } O(n)
\end{array}
$$

Although not efficient for these purposes, Haskell lists are frequently used as

- arrays
- queues, deques, catenable queues
- sets
- lookup tables, association lists, finite maps
- …

Why?

Other operations on lists

$$
\begin{array}{lll}
\mathit{snoc} & :: [a] \to a \to [a] & \text{-- } O\ (n) \\
\mathit{snoc} & = \lambda xs\ x \to xs \mathbin{+\!\!+} [x] & \\
(\mathbin{+\!\!+}) & :: [a] \to [a] \to [a] & \text{-- } O\ (n) \\
\mathit{reverse} & :: [a] \to [a] & \text{-- } O\ (n)\text{, naively: } O\ (n^2) \\
\mathit{union} & :: \mathsf{Eq}\ a \Rightarrow [a] \to [a] \to [a] & \text{-- } O\ (mn) \\
\mathit{elem} & :: \mathsf{Eq}\ a \Rightarrow a \to [a] \to \mathsf{Bool} & \text{-- } O\ (n)
\end{array}
$$

Although not efficient for these purposes, Haskell lists are frequently used as

- arrays
- queues, deques, catenable queues
- sets
- lookup tables, association lists, finite maps
- ...

Why?

Lists are everywhere, because ...

- There is a convenient built-in notation for lists.
- There are even list comprehensions in Haskell.
- Lots of library functions on lists.
- Pattern matching!
- Haskell strings are lists.
- Other data structures not widely known.
- Arrays are often worse.
- Not enough standard libraries for data structures.

We are going to change this ...

Unfortunately, the remaining reasons are valid.

Lists are everywhere, because . . .

- There is a convenient built-in notation for lists.
- There are even list comprehensions in Haskell.
- Lots of library functions on lists.
- Pattern matching!
- Haskell strings are lists.
- Other data structures not widely known.
- Arrays are often worse.
- Not enough standard libraries for data structures.

We are going to change this . . .

Unfortunately, the remaining reasons are valid.

Lists are everywhere, because ...

- There is a convenient built-in notation for lists.
- There are even list comprehensions in Haskell.
- Lots of library functions on lists.
- Pattern matching!
- Haskell strings are lists.
- Other data structures not widely known.
- Arrays are often worse.
- Not enough standard libraries for data structures.

We are going to change this ...

Unfortunately, the remaining reasons are valid.

Lists are everywhere, because . . .

- There is a convenient built-in notation for lists.
- There are even list comprehensions in Haskell.
- Lots of library functions on lists.
- Pattern matching!
- Haskell strings are lists.
- Other data structures not widely known.
- Arrays are often worse.
- Not enough standard libraries for data structures.

We are going to change this . . .

Unfortunately, the remaining reasons are valid.

Therefore, a bit of advice

Use accumulating arguments.

$$
\begin{aligned}
&\mathit{reverse} && :: [a] \rightarrow [a] \\
&\mathit{reverse} && = \mathit{reverse}'\ [\,] \\
&\mathit{reverse}' && :: [a] \rightarrow [a] \rightarrow [a] \\
&\mathit{reverse}'\ \mathit{acc}\ [\,] && = \mathit{acc} \\
&\mathit{reverse}'\ \mathit{acc}\ (x : \mathit{xs}) && = \mathit{reverse}'\ (x : \mathit{acc})\ \mathit{xs}
\end{aligned}
$$

The concatenation trick.

Compare:

$$
\begin{aligned}
&((((xs_1 +\!\!+) \cdot (xs_2 +\!\!+)) \cdot (xs_3 +\!\!+)) \cdot (xs_4 +\!\!+))\ ""\\
&(((xs_1 +\!\!+ xs_2) +\!\!+ xs_3) +\!\!+ xs_4)
\end{aligned}
$$

Therefore, a bit of advice

Use accumulating arguments.

$$
\begin{aligned}
&\mathit{reverse} &&:: [a] \to [a] \\
&\mathit{reverse} &&= \mathit{reverse}'\ [\,] \\
&\mathit{reverse}' &&:: [a] \to [a] \to [a] \\
&\mathit{reverse}'\ \mathit{acc}\ [\,] &&= \mathit{acc} \\
&\mathit{reverse}'\ \mathit{acc}\ (x : \mathit{xs}) &&= \mathit{reverse}'\ (x : \mathit{acc})\ \mathit{xs}
\end{aligned}
$$

The concatenation trick.

Compare:

$$
\begin{aligned}
&((((\mathit{xs}_1 +\!\!+) \cdot (\mathit{xs}_2 +\!\!+)) \cdot (\mathit{xs}_3 +\!\!+)) \cdot (\mathit{xs}_4 +\!\!+))\ \texttt{""} \\
&(((\mathit{xs}_1 +\!\!+ \mathit{xs}_2) +\!\!+ \mathit{xs}_3) +\!\!+ \mathit{xs}_4)
\end{aligned}
$$

Overview

About arrays

Imperative arrays feature

- constant-time lookup
- constant-time update

Update is usually at least as important as lookup.

Functional arrays do

- lookup in $O(1)$; yay!
- update in $O(n)$! Why? Persistence!

Array update is even worse than list update.

- To update the nth element of a list, $n - 1$ elements are copied.
- **To update any element of an array, the whole array is copied.**

About arrays

Imperative arrays feature

- constant-time lookup
- constant-time update

Update is usually at least as important as lookup.

Functional arrays do

- lookup in $O(1)$; yay!
- update in $O(n)$! Why? Persistence!

Array update is even worse than list update.

- To update the nth element of a list, $n - 1$ elements are copied.
- **To update any element of an array, the whole array is copied.**

About arrays

Imperative arrays feature

- constant-time lookup
- constant-time update

Update is usually at least as important as lookup.

Functional arrays do

- lookup in $O(1)$; yay!
- update in $O(n)$! Why? Persistence!

Array update is even worse than list update.

- To update the nth element of a list, $n-1$ elements are copied.
- **To update any element of an array, the whole array is copied.**

Space efficiency vs. space leaks

Arrays can be stored in a compact way.
Lists require lots of pointers.

If arrays are updated frequently and used persistently,
space leaks will occur!

Space efficiency vs. space leaks

Arrays can be stored in a compact way.
Lists require lots of pointers.

If arrays are updated frequently and used persistently,
space leaks will occur!

Mutable arrays

- Are like imperative arrays.
- Defined in Data.Array.MArray and Data.Array.IO.
- All operations in a state monad (possibly IO monad).
- Often awkward to use in a functional setting.

Overview

Introduction (Lists)

Arrays

Unboxed types

Queues and deques

Summary and next lecture

Boxed vs. unboxed types

Haskell data structures are boxed.

- Each value is behind an additional indirection.
- This allows polymorphic datastructures (because the size of a pointer is always the same).
- This allows laziness, because the pointer can be to a computation as well as to evaluated data.

GHC offers unboxed datatypes, too. Naturally, they

- are slightly more efficient (in both space and time),
- are strict,
- cannot be used in polymorphic data structures.

Boxed vs. unboxed types

Haskell data structures are boxed.

- Each value is behind an additional indirection.
- This allows polymorphic datastructures (because the size of a pointer is always the same).
- This allows laziness, because the pointer can be to a computation as well as to evaluated data.

GHC offers unboxed datatypes, too. Naturally, they

- are slightly more efficient (in both space and time),
- are strict,
- cannot be used in polymorphic data structures.

Unboxed types

- Defined in GHC.Base.
- For example, Int#, Char#, Double#.
- Have kind #, not $*$.
- Use specialized operations such as

  > (+#) :: Int# → Int# → Int#

- Cannot be used in polymorphic functions or datatypes.
- Are used by GHC internally to define the usual datatypes:

  > **data** Int = *I#* Int#

Packed strings

- Defined in Data.PackedString.
- Implemented as immutable, unboxed arrays.
- Can be more space-efficient than standard strings.
- Manipulating packed strings can be expensive.

Overview

Universiteit Utrecht

Queues

- Stacks are LIFO (last-in-first-out).
- Queues are FIFO (first-in-first-out).
- A list is not very suitable to represent a queue, because efficient access to both ends is desired.

The standard trick is:

$$\textbf{data}\ \text{Queue}\ a = Q\ [a]\ [a]$$

The first list is the front, the second the back of the queue, in reversed order.

Queues

- Stacks are LIFO (last-in-first-out).
- Queues are FIFO (first-in-first-out).
- A list is not very suitable to represent a queue, because efficient access to both ends is desired.

The standard trick is:

data Queue $a = Q\ [a]\ [a]$

The first list is the front, the second the back of the queue, in reversed order.

Queues

- Stacks are LIFO (last-in-first-out).
- Queues are FIFO (first-in-first-out).
- A list is not very suitable to represent a queue, because efficient access to both ends is desired.

The standard trick is:

$$\textbf{data}\ \text{Queue}\ a = Q\ [a]\ [a]$$

The first list is the front, the second the back of the queue, in reversed order.

Queue operations

This is what we want for a queue:

$$
\begin{array}{lll}
\mathit{empty} & :: \mathsf{Queue}\ a & \text{-- produce an empty queue} \\
\mathit{snoc} & :: a \rightarrow \mathsf{Queue}\ a \rightarrow \mathsf{Queue}\ a & \text{-- insert at the back} \\
\mathit{head} & :: \mathsf{Queue}\ a \rightarrow a & \text{-- get first element} \\
\mathit{tail} & :: \mathsf{Queue}\ a \rightarrow \mathsf{Queue}\ a & \text{-- remove first element}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{toList} & :: \mathsf{Queue}\ a \rightarrow [a] & \text{-- queue to list} \\
\mathit{fromList} & :: [a] \rightarrow \mathsf{Queue}\ a & \text{-- list to queue}
\end{array}
$$

Queue operations

This is what we want for a queue:

```
empty    :: Queue a                     -- produce an empty queue
snoc     :: a → Queue a → Queue a       -- insert at the back
head     :: Queue a → a                 -- get first element
tail     :: Queue a → Queue a           -- remove first element
```

```
toList   :: Queue a → [a]               -- queue to list
fromList :: [a] → Queue a               -- list to queue
```

Implementing queue operations

$$
\begin{aligned}
&\mathit{empty} :: \mathsf{Queue}\ a \\
&\mathit{empty} = Q\ [\,]\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{snoc} :: a \rightarrow \mathsf{Queue}\ a \rightarrow \mathsf{Queue}\ a \\
&\mathit{snoc}\ \ x\ \ (Q\ \mathit{fs}\ \mathit{bs}) = Q\ \mathit{fs}\ (x : \mathit{bs})
\end{aligned}
$$

Implementing queue operations

$$
\begin{aligned}
&\mathit{empty} :: \mathsf{Queue}\ a \\
&\mathit{empty} = Q\ [\,]\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{snoc} :: a \rightarrow \mathsf{Queue}\ a \rightarrow \mathsf{Queue}\ a \\
&\mathit{snoc}\ \ x\ \ (Q\ \mathit{fs}\ \mathit{bs}) = Q\ \mathit{fs}\ (x : \mathit{bs})
\end{aligned}
$$

Invocations of *reverse*

$$
\begin{aligned}
&\mathit{head} :: \mathsf{Queue}\ a && \to a\\
&\mathit{head}\ \ (Q\ (f : \mathit{fs})\ \mathit{bs}) && = f\\
&\mathit{head}\ \ (Q\ [\,]\ \mathit{bs}) && = \mathit{head}\ (\mathit{reverse}\ \mathit{bs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{tail} :: \mathsf{Queue}\ a && \to \mathsf{Queue}\ a\\
&\mathit{tail}\ \ (Q\ (f : \mathit{fs})\ \mathit{bs}) && = Q\ \mathit{fs}\ \mathit{bs}\\
&\mathit{tail}\ \ (Q\ [\,]\ \mathit{bs}) && = Q\ (\mathit{tail}\ (\mathit{reverse}\ \mathit{bs}))\ [\,]
\end{aligned}
$$

Persistence spoils the fun here; without persistence, all operations would be in $O(1)$ amortized time.

Invocations of *reverse*

$$\begin{aligned}
&\mathit{head} :: \mathsf{Queue}\ a &&\to a\\
&\mathit{head}\ \ (Q\ (f : \mathit{fs})\ \mathit{bs}) &&= f\\
&\mathit{head}\ \ (Q\ [\,]\ \qquad \mathit{bs}) &&= \mathit{head}\ (\mathit{reverse}\ \mathit{bs})
\end{aligned}$$

$$\begin{aligned}
&\mathit{tail} :: \mathsf{Queue}\ a &&\to \mathsf{Queue}\ a\\
&\mathit{tail}\ \ (Q\ (f : \mathit{fs})\ \mathit{bs}) &&= Q\ \mathit{fs}\ \mathit{bs}\\
&\mathit{tail}\ \ (Q\ [\,]\ \qquad \mathit{bs}) &&= Q\ (\mathit{tail}\ (\mathit{reverse}\ \mathit{bs}))\ [\,]
\end{aligned}$$

Persistence spoils the fun here; without persistence, all operations would be in $O(1)$ amortized time.

Invocations of *reverse*

$$\begin{array}{llll} head :: \text{Queue } a & & \rightarrow & a \\ head & (Q\ (f : fs)\ bs) & = & f \\ head & (Q\ [\,]\ \ \ \ \ \ \ \ bs) & = & head\ (reverse\ bs) \end{array}$$

$$\begin{array}{llll} tail :: \text{Queue } a & & \rightarrow & \text{Queue } a \\ tail & (Q\ (f : fs)\ bs) & = & Q\ fs\ bs \\ tail & (Q\ [\,]\ \ \ \ \ \ \ \ bs) & = & Q\ (tail\ (reverse\ bs))\ [\,] \end{array}$$

Persistence spoils the fun here; without persistence, all operations would be in $O(1)$ amortized time.

Amortized analysis

Amortized complexity can be better than worst-case complexity if the worst-case cannot occur that often in practice.

In an amortized analysis, we look at the cost of multiple operations rather than single operations.

Idea

- Distribute the work that *reverse* causes over multiple operations in such a way that the amortized cost of each operation is constant.
- Use laziness (and memoization) to ensure that expensive operations are not performed too early or too often.

Memoization

A suspended expression in a lazy language is evaluated only once. The suspension is then updated with the result. Whenever the same expression is needed again, the result can be used immediately.

This is called memoization.

Memoization

A suspended expression in a lazy language is evaluated only once. The suspension is then updated with the result. Whenever the same expression is needed again, the result can be used immediately.

This is called memoization.

Efficient queues

Recall the queue representation:

> **data** Queue $a = Q \ [a] \ [a]$

As we will see, the work of reversing the list can be distributed well by chosing the following invariant for $Q \ fs \ bs$:

> $\mathit{length}\ \mathit{fs} \geqslant \mathit{length}\ \mathit{bs}$

In particular, $\mathit{length}\ \mathit{fs} == 0$ if and only if the queue is empty.

We need the lengths of both lists available in constant time.

Efficient queues

Recall the queue representation:

$$\textbf{data}\ \mathsf{Queue}\ a = Q\ \ [a]\ \ [a]$$

As we will see, the work of reversing the list can be distributed well by chosing the following invariant for $Q\ \ fs\ \ bs$:

$$\mathit{length}\ \mathit{fs} \geqslant \mathit{length}\ \mathit{bs}$$

In particular, $\mathit{length}\ \mathit{fs} == 0$ if and only if the queue is empty.

We need the lengths of both lists available in constant time.

Efficient queues

Recall the queue representation:

$\mathbf{data}\ \mathsf{Queue}\ a = Q\ \ !\,\mathsf{Int}\ [a]\ \ !\,\mathsf{Int}\ [a]$

As we will see, the work of reversing the list can be distributed well by chosing the following invariant for $Q\ \mathit{lf}\ \mathit{fs}\ \mathit{lb}\ \mathit{bs}$:

$\mathit{lf} \geqslant \mathit{lb}$

In particular, $\mathit{length}\ \mathit{fs} == 0$ if and only if the queue is empty.

We need the lengths of both lists available in constant time.

empty and *head* are simple due to the invariant

$$
\begin{aligned}
&\mathit{empty} :: \mathsf{Queue}\ a \\
&\mathit{empty} = Q\ 0\ [\,]\ 0\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{head} :: \mathsf{Queue}\ a && \to a \\
&\mathit{head}\ \ (Q\ \_\ (f : fs)\ bs) && = f \\
&\mathit{head}\ \ (Q\ \_\ [\,]\ \ \_) && = \mathit{error}\ \texttt{"empty queue"}
\end{aligned}
$$

empty and *head* are simple due to the invariant

$$
\begin{aligned}
&\mathit{empty} :: \mathsf{Queue}\ a \\
&\mathit{empty} = Q\ 0\ [\,]\ 0\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{head} :: \mathsf{Queue}\ a \rightarrow a \\
&\mathit{head}\ (Q\ \_\ (f : \mathit{fs})\ \mathit{bs}) = f \\
&\mathit{head}\ (Q\ \_\ [\,]\ \_) = \mathit{error}\ \texttt{"empty queue"}
\end{aligned}
$$

What about *tail* and *snoc*?

$$
\begin{array}{llcl}
\mathit{tail} :: \mathsf{Queue}\ a & & \rightarrow & a \\
\mathit{tail} & (Q\ \mathit{lf}\ (f : \mathit{fs})\ \mathit{lb}\ b) & = & \mathit{makeQ}\ (\mathit{lf} - 1)\ \mathit{fs}\ \mathit{lb}\ b \\
\mathit{tail} & (Q\ \_\ [\,]\ \_\ \_) & = & \mathit{error}\ \texttt{"empty queue"}
\end{array}
$$

$$
\begin{array}{lllcl}
\mathit{snoc} :: a \rightarrow \mathsf{Queue}\ a & & & \rightarrow & \mathsf{Queue}\ a \\
\mathit{snoc} & x & (Q\ \mathit{lf}\ \mathit{fs}\ \mathit{lb}\ \mathit{bs}) & = & \mathit{makeQ}\ \mathit{lf}\ \mathit{fs}\ (\mathit{lb} + 1)\ (x : \mathit{bs})
\end{array}
$$

In both cases, we have to make a new queue using a call $\mathit{makeQ}\ \mathit{lf}\ f\ \mathit{lb}\ f'$, where we may need to re-establish the invariant.

What about *tail* and *snoc*?

$$
\begin{aligned}
&\mathit{tail} :: \mathsf{Queue}\ a && \to a \\
&\mathit{tail}\ \ (Q\ \mathit{lf}\ (f : \mathit{fs})\ \mathit{lb}\ b) && = \mathit{makeQ}\ (\mathit{lf} - 1)\ \mathit{fs}\ \mathit{lb}\ b \\
&\mathit{tail}\ \ (Q\ \_\ [\,]\ \_\ \_) && = \mathit{error}\ \texttt{"empty queue"}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{snoc} :: a \to \mathsf{Queue}\ a && \to \mathsf{Queue}\ a \\
&\mathit{snoc}\ \ x\ \ (Q\ \mathit{lf}\ \mathit{fs}\ \mathit{lb}\ \mathit{bs}) && = \mathit{makeQ}\ \mathit{lf}\ \mathit{fs}\ (\mathit{lb} + 1)\ (x : \mathit{bs})
\end{aligned}
$$

In both cases, we have to make a new queue using a call $\mathit{makeQ}\ \mathit{lf}\ f\ \mathit{lb}\ f'$, where we may need to re-establish the invariant.

What about *tail* and *snoc*?

$$
\begin{aligned}
&\mathit{tail} :: \mathsf{Queue}\ a && \rightarrow a \\
&\mathit{tail}\ \ (Q\ \mathit{lf}\ (f : \mathit{fs})\ \mathit{lb}\ b) && = \mathit{makeQ}\ (\mathit{lf} - 1)\ \mathit{fs}\ \mathit{lb}\ b \\
&\mathit{tail}\ \ (Q\ \_\ [\,]\ \_\ \_) && = \mathit{error}\ \texttt{"empty queue"}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{snoc} :: a \rightarrow \mathsf{Queue}\ a && \rightarrow \mathsf{Queue}\ a \\
&\mathit{snoc}\ \ x\ \ (Q\ \mathit{lf}\ \mathit{fs}\ \mathit{lb}\ \mathit{bs}) && = \mathit{makeQ}\ \mathit{lf}\ \mathit{fs}\ (\mathit{lb} + 1)\ (x : \mathit{bs})
\end{aligned}
$$

In both cases, we have to make a new queue using a call $\mathit{makeQ}\ \mathit{lf}\ f\ \mathit{lb}\ f'$, where we may need to re-establish the invariant.

How to make a queue

$$
\begin{array}{l}
\mathit{makeQ} :: \mathsf{Int} \to [\,a\,] \to \mathsf{Int} \to [\,a\,] \\
\qquad\qquad\qquad\quad \to \mathsf{Queue}\ a \\
\mathit{makeQ} \quad \mathit{lf} \qquad \mathit{fs} \qquad \mathit{lb} \qquad \mathit{bs} \\
\quad \mid \mathit{lf} \geqslant \mathit{lb} \qquad = Q\ \mathit{lf}\ \mathit{fs}\ \mathit{lb}\ \mathit{bs} \\
\quad \mid \mathit{otherwise} = Q\ (\mathit{lf} + \mathit{lb})\ (\mathit{fs} +\!\!+\ \mathit{reverse}\ \mathit{bs})\ 0\ [\,]
\end{array}
$$

Why is this implementation “better”?

(drawing and lots of handwaving)

Read Okasaki’s book for a proof.

Why is this implementation "better"?

(drawing and lots of handwaving)

Read Okasaki's book for a proof.

Queues in GHC

- Available in Data.Queue (ghc-6.4).
- Based on a slight variation of the implementation described here, allowing operations in constant worse-case time ("real-time queues").
- Representation of queues is then

  data Queue $a = Q\ [a]\ [a]\ [a]$

  where the third list is used to maintain the unevaluated part of the front queue.
- Described in the paper *Simple and efficient functional queues and deques*, JFP 5(4), pages 583–592, October 1995, by Chris Okasaki.
- Also described in Okasaki's book.

Queues in GHC

- Available in Data.Queue (ghc-6.4).
- Based on a slight variation of the implementation described here, allowing operations in constant worse-case time (“real-time queues”).
- Representation of queues is then

  $\textbf{data}\ \mathsf{Queue}\ a = Q\ [a]\ [a]\ [a]$

  where the third list is used to maintain the unevaluated part of the front queue.
- Described in the paper *Simple and efficient functional queues and deques*, JFP 5(4), pages 583–592, October 1995, by Chris Okasaki.
- Also described in Okasaki’s book.

Queues in GHC

- Available in Data.Queue (ghc-6.4).
- Based on a slight variation of the implementation described here, allowing operations in constant worse-case time ("real-time queues").
- Representation of queues is then

  $$\textbf{data}\ \mathsf{Queue}\ a = Q\ [a]\ [a]\ [a]$$

  where the third list is used to maintain the unevaluated part of the front queue.
- Described in the paper *Simple and efficient functional queues and deques*, JFP 5(4), pages 583–592, October 1995, by Chris Okasaki.
- Also described in Okasaki's book.

Deques

A deque is a double-ended queue.

Operations like queue operations:

$$
\begin{array}{lll}
\mathit{empty} & :: \text{Deque } a & \text{-- produce an empty queue} \\
\mathit{snoc} & :: a \rightarrow \text{Deque } a \rightarrow \text{Deque } a & \text{-- insert at the back} \\
\mathit{head} & :: \text{Deque } a \rightarrow a & \text{-- get first element} \\
\mathit{tail} & :: \text{Deque } a \rightarrow \text{Deque } a & \text{-- remove first element} \\
\mathit{toList} & :: \text{Deque } a \rightarrow [a] & \text{-- queue to list} \\
\mathit{fromList} & :: [a] \rightarrow \text{Deque } a & \text{-- list to queue}
\end{array}
$$

Additionally (also in constant time):

$$
\begin{array}{lll}
\mathit{cons} & :: a \rightarrow \text{Deque } a \rightarrow \text{Deque } a & \text{-- insert at the front} \\
\mathit{init} & :: \text{Deque } a \rightarrow \text{Deque } a & \text{-- remove last element} \\
\mathit{last} & :: \text{Deque } a \rightarrow a & \text{-- get last element}
\end{array}
$$

Deques

A deque is a double-ended queue.

Operations like queue operations:

```
empty    :: Deque a                     -- produce an empty queue
snoc     :: a → Deque a → Deque a       -- insert at the back
head     :: Deque a → a                 -- get first element
tail     :: Deque a → Deque a           -- remove first element
toList   :: Deque a → [a]               -- queue to list
fromList :: [a] → Deque a               -- list to queue
```

Additionally (also in constant time):

```
cons :: a → Deque a → Deque a   -- insert at the front
init :: Deque a → Deque a       -- remove last element
last :: Deque a → a             -- get last element
```

Efficient deques

The previous implementation can easily be extended to work for deques:

$$\textbf{data}\ \mathsf{Deque}\ a = D\ !\,\mathsf{Int}\ [a]\ !\,\mathsf{Int}\ [a]$$

Of course, we have to make the representation more symmetric. The invariant for $D\ lf\ fs\ lb\ bs$ becomes:

$$lf \leqslant c * lb + 1 \wedge lr \leqslant c * lf + 1$$

(for some constant $c > 1$).

Implementation of deques

- The implementation of *makeQ* must be adapted to maintain this invariant.
- The other operations are straight-forward, we only have to pay attention to the one-element queue.
- How much time does it cost to reverse a deque?
- Unfortunately, there currently is no standard Haskell library for deques.

Catenable queues or deques

- Queues or deques that support efficient concatenation are called catenable.
- It is possible to support concatenation in $O(\log n)$ and even in $O(1)$ amortized time, but this requires a completely different implementation of queues/deques.
- Again, there currently are no standard Haskell libraries for catenable queues and deques.

Overview

Introduction (Lists)

Arrays

Unboxed types

Queues and deques

Summary and next lecture

Summary

- Lists are everwhere in Haskell, for a lot of good reasons.
- Functional data structures are persistent.
- Persistence and efficiency and evaluation order all interact.
- Array updates are inherently inefficient in a functional language.
- Queues and deques support many operations efficiently that normal lists do not.
- In a persistent setting, queue and deque operations can be implemented with the same complexity bounds as in an ephemeral setting.
- GHC has a standard library that supports many (but not all desirable) datastructures, for instance lists, queues, arrays in all flavors, but also unboxed types and packed strings.

Next lecture

- Pattern matching, abstract datatypes, views.
- Trees, finite maps and sets.
- . . .