Advanced Functional Programming (7)
Functional Data Structures

Bastiaan Heeren

Center for Software Technology, Universiteit Utrecht
http://www.cs.uu.nl/groups/ST/

February 21, 2007

(slides are based on a guest lecture by Andres Löh)

Overview

1. Introduction
2. Arrays
3. Unboxed types
4. Queues
5. Summary

Question

What is the most frequently used data structure in Haskell?

Question

What is the most frequently used data structure in Haskell?

Clearly lists.....what are lists good for?

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 – all operate in *O(1)*
- These are the *stack* operations (*top*, *pop*, and *push*)

Haskell stacks are persistent

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

Haskell stacks are persistent

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

If not persistent, a data structure is called *ephemeral*.

Haskell stacks are persistent

A data structure is called *persistent* if after an operation both the original and the resulting 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

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

Other operations on lists

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

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..

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

Lists are everywhere because..

1. There is a convenient built-in notation for lists.
2. There are even list comprehensions in Haskell!
3. Lots of library functions on lists.
4. Pattern matching!
5. Haskell strings are lists.
6. Other data structures are not widely known.
7. Arrays are often worse.
8. Not enough standard libraries for data structures
   $\Rightarrow$ *Fortunately, this is changing!*

Lists are everywhere because..

1. There is a convenient built-in notation for lists.
2. There are even list comprehensions in Haskell!
3. Lots of library functions on lists.
4. Pattern matching!
5. Haskell strings are lists.
6. Other data structures are not widely known.
7. Arrays are often worse.
8. Not enough standard libraries for data structures
   $\Rightarrow$ *Fortunately, this is changing!*

We are going to change item 6: the remaining reasons are still valid.

Therefore, a bit of advice

- Use accumulating arguments.

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

Therefore, a bit of advice

- Use accumulating arguments.

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

- Use the concatenation trick.
  Compare

$$((\mathit{xs}_1 +\!\!+) \circ (\mathit{xs}_2 +\!\!+) \circ (\mathit{xs}_3 +\!\!+) \circ (\mathit{xs}_4 +\!\!+))\ [\,]$$

  with

$$((\mathit{xs}_1 +\!\!+ \mathit{xs}_2) +\!\!+ \mathit{xs}_3) +\!\!+ \mathit{xs}_4$$

About arrays

Imperative arrays feature:

- constant-time lookup
- constant-time update

Update is usually at least as important as lookup.

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 $O(1)$; nice.
- update in $O(n)$; not so nice.

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 $O(1)$; nice.
- update in $O(n)$; not so nice. $\Rightarrow$ Why? Persistence!

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 $O(1)$; nice.
- update in $O(n)$; not so nice. $\Rightarrow$ Why? Persistence!

Array update is even worse than list update.

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

Space efficiency versus 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 versus 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.

Boxed versus 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, and
- cannot be used in polymorphic data structures.

Unboxed types

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

$$(+\#) :: Int\# \rightarrow Int\# \rightarrow Int\#$$

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

$$\textbf{data}\ Int = I\#\ Int\#$$

Byte strings

A space-efficient representation of a Word8 vector, supporting many efficient operations.

- Defined in Data.ByteString
- Previously known as Packed strings (depricated)
- Uses low-level mutable arrays in C
- Provides a purely functional interface
- Performance boost due to equational transformations (fusion strategies)
- C-like speed while retaining idiomatic Haskell brevity and clarity

Unfortunately, byte strings are restricted to *Word8*, although they could be polymorphic in the type of elements.

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.

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

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

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:

```
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
```

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
```

Two additional functions

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

Implementing queue operations

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

Implementing queue operations

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

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

Invocations of *reverse*

Before we can define *head* and *tail* on queues:

```
import Prelude hiding (head, tail)
import qualified Prelude as P
```

Invocations of *reverse*

Before we can define *head* and *tail* on queues:

```
import Prelude hiding (head, tail)
import qualified Prelude as P
```

```
head :: Queue a → a
head (Q (f : fs) bs) = f
head (Q []       bs) = P.head (reverse bs)
```

Invocations of *reverse*

Before we can define *head* and *tail* on queues:

```
import Prelude hiding (head, tail)
import qualified Prelude as P
```

```
head :: Queue a → a
head (Q (f : fs) bs) = f
head (Q []       bs) = P.head (reverse bs)
```

```
tail :: Queue a → Queue a
tail (Q (f : fs) bs) = Q fs bs
tail (Q []       bs) = Q (P.tail (reverse bs)) []
```

Invocations of *reverse*

Before we can define *head* and *tail* on queues:

```
import Prelude hiding (head, tail)
import qualified Prelude as P
```

```
head :: Queue a → a
head (Q (f : fs) bs) = f
head (Q []       bs) = P.head (reverse bs)
```

```
tail :: Queue a → Queue a
tail (Q (f : fs) bs) = Q fs bs
tail (Q []       bs) = Q (P.tail (reverse bs)) []
```

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 operations is constant.
- Use laziness (and memoization) to ensure that expensive operations are not performed too early or too often.

Idea

- Distribute the work that *reverse* causes over multiple operations in such a way that the amortized cost of each operations 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*.

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 choosing the following invariant for $Q\ fs\ bs$:

$$length\ fs \geqslant length\ bs$$

In particular, $length\ fs \equiv 0$ if and only if the queue is empty.

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 choosing the following invariant for *Q fs bs*:

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

In particular, *length fs* $\equiv 0$ if and only if the queue is empty. We need the lengths of both lists available in constant time:

New definition:

data *Queue a* $=$ *Q* !*Int* $[a]$!*Int* $[a]$

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 choosing the following invariant for *Q fs bs*:

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

In particular, *length fs* $\equiv 0$ if and only if the queue is empty. We need the lengths of both lists available in constant time:

New definition:

data *Queue a* $=$ *Q* !*Int* $[a]$!*Int* $[a]$

A !*Int* is a *strict* value of type *Int*.

Implementing queue operations

empty and *head* are simple due to the invariant

$$\mathit{empty} :: \mathit{Queue}\ a$$
$$\mathit{empty} = Q\ 0\ [\,]\ 0\ [\,]$$

Implementing queue operations

empty and *head* are simple due to the invariant

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

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

What about *tail* and *snoc*?

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

What about *tail* and *snoc*?

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

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

What about *tail* and *snoc*?

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

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

In both cases, we have to make a new queue using a call to *makeQ lf fs lb bs*, where we may need to re-establish the invariant.

makeQ is called a *smart constructor*: besides constructing a queue, it maintains our invariant.

How to make a queue

Our smart constructor is defined as follows:

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

How to make a queue

Our smart constructor is defined as follows:

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

Question: Why is this implementation “better”?

How to make a queue

Our smart constructor is defined as follows:

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

Question: 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 and later)
- 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

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

  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 (double-ended queues)

Operations like queue operations

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

Deques (double-ended queues)

Operations like queue operations

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

Additionally (also in constant time)

cons	$:: a \rightarrow \mathit{Deque}\ a \rightarrow \mathit{Deque}\ a$	-- insert at the front
init	$:: \mathit{Deque}\ a \rightarrow \mathit{Deque}\ a$	-- remove last element
last	$:: \mathit{Deque}\ a \rightarrow a$	-- get last element

Efficient deques

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

data *Deque a* $=$ *D* !*Int* [*a*] !*Int* [*a*]

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

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

for some constant $C > 1$.

Implementation of deques

- The implementation of *makeQ* must be adapted to maintain this invariant.
- The other operations are straightforward, we only have to pay attention to the one-element queue.
- How much time does it cost to reverse a deque?
- Data.Sequence implements these operations using 2-3 finger trees.

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.
- Data.Sequence implements concatenation in $O\ (log\ (min\ (n_1, n_2)))$

Summary

- Lists are everywhere 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) data structures, for instance lists, queues, arrays in all flavors, but also unboxed types and packed strings.

Agenda

1. Read *"Finger trees: a simple general-purpose data structure (Ralf Hinze and Ross Paterson)".*
2. This Friday: deadline Exercise 1 (state monad).
3. Ant team have to make an appointment with me for next week.

Next lecture:

February 26 (Monday): Functional Data Structures Part II.