

Advanced Functional Programming 2011-2012, period 2

Andres Löh and Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

Dec 1, 2011

6. Data structures



Question

What is the most frequently used data structure in Haskell?

Question

What is the most frequently used data structure in Haskell?

Lists, clearly ...

6.1 Lists everywhere





head :: $[a] \rightarrow a$ tail :: $[a] \rightarrow [a]$ (:) :: $a \rightarrow [a] \rightarrow [a]$

◆ロト ◆昼 ト ◆ 差 ト → 差 り へ ○

head ::
$$[a] \rightarrow a$$

tail :: $[a] \rightarrow [a]$
(:) :: $a \rightarrow [a] \rightarrow [a]$

► These are efficient operations on lists.

► These are efficient operations on lists.

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

- ► These are efficient operations on lists.
- ► These are the stack operations.
- ► Haskell lists are persistent stacks.

Persistence

- ▶ 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.

Persistence

- ▶ 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.
- Persistence can have an effect on the efficiency of data structures.



Other operations on lists

Other operations on lists

Often, Haskell lists are used inefficiently as

- arrays
- queues, double-ended queues, catenable queues
- sets, lookup tables, association lists, finite maps





Why?

- Special language support for lists:
 - There is a convenient built-in notation for lists.
 - List comprehensions.
 - Pattern matching.
- Libraries:
 - Lots of library functions on lists.
 - ▶ In the past, there were few standard libraries for data structures.
- Lack of knowledge:
 - Lists are easy to learn, but where are the other data structures?
 - Just switching from lists to arrays can make matters worse.

Why?

- Special language support for lists:
 - ▶ There is a convenient built-in notation for lists.
 - List comprehensions.
 - Pattern matching.
- Libraries:
 - Lots of library functions on lists.
 - In the past, there were few standard libraries for data structures.
- ► Lack of knowledge:
 - Lists are easy to learn, but where are the other data structures?
 - ▶ Just switching from lists to arrays can make matters worse.

Language support is best for lists, but other data structures are reasonably easy to use as well.

6.2 Arrays





Imperative vs. functional arrays

Imperative (mutable) arrays

- constant-time lookup
- constant-time update
- are ephemeral

Update is usually at least as important as lookup.

Imperative vs. functional arrays

Imperative (mutable) arrays

- ► constant-time lookup
- constant-time update
- are ephemeral

Update is usually at least as important as lookup.

Functional (immutable) arrays

- ► available in Data.Array
- ▶ lookup in O(1); yay!
- ▶ update in O(n)!



Imperative vs. functional arrays

Imperative (mutable) arrays

- constant-time lookup
- constant-time update
- are ephemeral

Update is usually at least as important as lookup.

Functional (immutable) arrays

- available in Data.Array
- ▶ lookup in O(1); yay!
- update in O(n)!
- Why? Persistence!





Array update vs. list update

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.
- ▶ Update of functional arrays is slow.
- ▶ If functional arrays are updated frequently and used persistently, space leaks will occur!

Mutable arrays

- ► Are like imperative arrays.
- Defined in Data.Array.IO (or Data.Array.ST).
- All operations in IO (or ST).
- Often awkward to use in a functional setting.
- ► Can be useful if you do not need persistence, but require frequent updates.



Interface of immutable arrays

Data.Array

```
data Array i e -- abstract
-- creation
array :: (lx i) \Rightarrow (i,i) \rightarrow [(i,e)] \rightarrow Array i e
listArray :: (lx i) \Rightarrow (i,i) \rightarrow [e] \rightarrow Array i e
-- lookup
(!) :: (lx i) \Rightarrow Array i e \rightarrow i \rightarrow e
bounds :: (lx i) \Rightarrow Array i e \rightarrow (i,i)
-- update
(//) :: (lx i) \Rightarrow Array i e \rightarrow [(i,e)] \rightarrow Array i e
-- destruction
          -- destruction
elems :: (Ix i) \Rightarrow Array i e \rightarrow [e]
assocs :: (Ix i) \Rightarrow Array i e \rightarrow [(i, e)]
```

Interface of mutable arrays

Data.Array.IO

```
data IOArray i e -- abstract
 \mathsf{newArray} \quad :: (\mathsf{Ix}\;\mathsf{i}) \Rightarrow (\mathsf{i},\mathsf{i}) \rightarrow \mathsf{e} \rightarrow \mathsf{IO}\;(\mathsf{IOArray}\;\mathsf{i}\;\mathsf{e})
newListArray :: (Ix i) \Rightarrow (i, i) \rightarrow [e] \rightarrow IO (IOArray i e)
 \begin{array}{ll} \mathsf{readArray} & :: (\mathsf{Ix}\;\mathsf{i}) \Rightarrow \mathsf{IOArray}\;\mathsf{i}\;\mathsf{e} \to \mathsf{i} \to \mathsf{IO}\;\mathsf{e} \\ \mathsf{getBounds} & :: (\mathsf{Ix}\;\mathsf{i}) \Rightarrow \mathsf{IOArray}\;\mathsf{i}\;\mathsf{e} \to \mathsf{IO}\;(\mathsf{i},\mathsf{i}) \\ \end{array} 
  writeArray :: (Ix i) \Rightarrow IOArray i e \rightarrow i \rightarrow e \rightarrow IO ()
   \begin{array}{ll} \mathsf{getElems} & :: (\mathsf{Ix}\; \mathsf{i}) \Rightarrow \mathsf{Array}\; \mathsf{i}\; \mathsf{e} \to \mathsf{IO}\; [\mathsf{e}] \\ \mathsf{getAssocs} & :: (\mathsf{Ix}\; \mathsf{i}) \Rightarrow \mathsf{Array}\; \mathsf{i}\; \mathsf{e} \to \mathsf{IO}\; [(\mathsf{i},\mathsf{e})] \end{array}
```



Conversion

freeze :: (Ix i)
$$\Rightarrow$$
 IOArray i e \rightarrow IO (Array i e) thaw :: (Ix i) \Rightarrow Array i e \rightarrow IO (IOArray i e)

Diff arrays

Data.Array.Diff

- ► Same interface as immutable arrays, i.e., not tied to monadic code.
- ▶ Implemented using destructive updates, i.e, update is O(1).

◆□▶◆御▶◆団▶◆団▶ 団 めの◎

Unboxed arrays

Data.Array.Unboxed

- Only available for specific types: Bool, Char, Int, Float, Double, and a few others.
- Internally uses unboxed values.
- ▶ Allows efficient storage, no laziness.

6.3 Unboxed types



Unboxed types

 $\begin{array}{l} \mathsf{Prelude} \rangle \ : \mathsf{i} \ \mathsf{Char} \\ \textbf{data} \ \mathsf{Char} = \mathsf{GHC}.\mathsf{Base}.\mathsf{C\#} \ \mathsf{GHC}.\mathsf{Prim}.\mathsf{Char} \# \end{array}$

- ► Char# is the type of unboxed characters.
- Unboxed types are a GHC extension.
- Use the MagicHash language pragma.
- Import GHC.Exts.
- ▶ Unboxed types are not stored on the heap.
- ▶ No indirection, thus more efficient in space and time.
- Thus no laziness.
- Also no polymorphism.
- ▶ No polymorphism means no use of general data structures!



Fixed-size types

- ► The size of Int is not exactly specified in the Report (there's a minimum range it has to cover).
- ▶ Numbers of type Integer are unbounded.
- ► Haskell also provides datatypes for numbers (and characters) of exact size.
- Module Data.Int defines Int8, Int16, Int32 and Int64 for signed integers.
- Module Data.Word defines Word8, Word16, Word32 and Word64 for unsigned integers.
- These types are particularly useful when interfacing to other languages.
- ► These type are boxed by default, but have unboxed variants in GHC.



6.4 ByteStrings



Haskell strings

type String
$$=$$
 [Char]

- Recall how Haskell lists and characters are represented.
- ▶ Strings are convenient to use (lists, again), but quite space-inefficient.
- Could an array representation help?

An example

A function to compute a hash of all alphabetic characters in a file f:

$$\label{eq:continuous} \begin{split} \text{return} \circ & \text{foldI'} \text{ hash } 5381 \circ \text{map toLower} \circ \\ & \text{filter isAlpha} = \!\!\!\! \ll \text{readFile f} \\ & \text{where hash h c} = \text{h} * 33 + \text{ord c} \end{split}$$

$$(=<) :: (a \rightarrow IO b) \rightarrow IO a \rightarrow IO b$$

 $(>=) :: IO a \rightarrow (a \rightarrow IO b) \rightarrow IO b$

- ► Often, strings are used as streams the string is traversed, modified, and written possibly several times.
- How many times does this code traverse the string?
- ▶ How many copies of the string are made?

An example

A function to compute a hash of all alphabetic characters in a file f:

return \circ foldl' hash $5381 \circ$ map toLower \circ filter isAlpha $= \ll$ readFile f where hash h c = h * 33 + ord c

$$(=<) :: (a \rightarrow IO b) \rightarrow IO a \rightarrow IO b$$

 $(>=) :: IO a \rightarrow (a \rightarrow IO b) \rightarrow IO b$

- ► Often, strings are used as streams the string is traversed, modified, and written possibly several times.
- ▶ How many times does this code traverse the string?
- ▶ How many copies of the string are made?
- Optimization is highly desirable.

[Faculty of Science Information and Computing Sciences]



Universiteit Utrecht

ByteString

Data.ByteString

- Reimplements most list functions.
- Uses a compact representation as an array of (unboxed) characters.
- Makes use of array fusion to
 - decrease the number of traversals.
 - decrease the number of copies of the data structure.
- There is a lazy variant Data.ByteString.Lazy. (Why?)



Fusion and Deforestation

- ► The ability to merge multiple traversals of a data structure into a single traversal is called fusion.
- ► The elimination of intermediate data structures is often called deforestation.
- Well-studied theory for the case of lists.
- ► The "Rewriting Haskell Strings" paper presents a new way of array/stream fusion.

Deforestation

Basic idea: "If we have a function which returns a value of some data type over which we subsequently fold, then we can replace the **constructors used** in that function to build that result by the corresponding arguments of the call to foldr."

Excursion: unfoldr

Note that these **constructing sites** are well visible in the function unfold:

```
\begin{split} & \text{unfoldr} :: (\mathsf{s} \to \mathsf{Maybe}\ (\mathsf{a},\mathsf{s})) \to \mathsf{s} \to [\mathsf{a}] \\ & \text{unfoldr}\ \mathsf{next}\ \mathsf{s} = \\ & \textbf{case}\ \mathsf{next}\ \mathsf{s}\ \textbf{of} \\ & \mathsf{Nothing}\ \to [] \\ & \mathsf{Just}\ (\mathsf{x},\mathsf{r}) \to \mathsf{x} : \mathsf{unfoldr}\ \mathsf{next}\ \mathsf{r} \end{split}
```

Excursion: unfoldr

Note that these **constructing sites** are well visible in the function unfold:

```
\begin{array}{l} \text{unfoldr} :: (s \rightarrow \mathsf{Maybe}\ (\mathsf{a}, \mathsf{s})) \rightarrow \mathsf{s} \rightarrow [\mathsf{a}] \\ \text{unfoldr}\ \mathsf{next}\ \mathsf{s} = \\ \textbf{case}\ \mathsf{next}\ \mathsf{s}\ \textbf{of} \\ \mathsf{Nothing}\ \rightarrow [] \\ \mathsf{Just}\ (\mathsf{x}, \mathsf{r}) \rightarrow \mathsf{x} : \mathsf{unfoldr}\ \mathsf{next}\ \mathsf{r} \end{array}
```

```
\begin{array}{ll} \text{repeat} &= \text{unfoldr } (\lambda \mathsf{x} \to \mathsf{Just} \ (\mathsf{x},\mathsf{x})) \\ \text{replicate n } \mathsf{x} &= \text{unfoldr } (\lambda \mathsf{n} \to \mathsf{if} \ \mathsf{n} == 0 \ \mathsf{then} \ \mathsf{Nothing} \\ &\quad \quad \mathsf{else} \ \mathsf{Just} \ (\mathsf{x},\mathsf{n}-1)) \ \mathsf{n} \\ \text{enumFromTo b } \mathsf{e} &= \mathsf{unfoldr} \ (\lambda \mathsf{b} \to \mathsf{if} \ \mathsf{b} > \mathsf{e} \ \mathsf{then} \ \mathsf{Nothing} \\ &\quad \quad \mathsf{else} \ \mathsf{Just} \ (\mathsf{b},\mathsf{b}+1)) \ \mathsf{b} \end{array}
```

unfoldr vs. foldr

unfoldr vs. foldr

$$\begin{array}{l} \mathsf{foldr} :: (\mathsf{r}, \mathsf{a} \to \mathsf{r} \to \mathsf{r}) \to [\mathsf{a}] \to \mathsf{r} \\ \mathsf{foldr} :: ((): + : (\mathsf{a}, \mathsf{r}) \to \mathsf{r}) \to [\mathsf{a}] \to \mathsf{r} \end{array}$$

unfoldr vs. foldr

$$\begin{array}{ll} \text{unfoldr} :: (\mathsf{s} \to \mathsf{Maybe}\ (\mathsf{a},\mathsf{s})) \to \mathsf{s} \to [\mathsf{a}] \\ \\ \text{foldr} & :: (\mathsf{a} \to \mathsf{r} \to \mathsf{r}) \to \mathsf{r} \to [\mathsf{a}] \to \mathsf{r} \end{array}$$

foldr ::
$$(r, a \rightarrow r \rightarrow r) \rightarrow [a] \rightarrow r$$

foldr :: $((): +: (a, r) \rightarrow r) \rightarrow [a] \rightarrow r$

Maybe a
$$\approx$$
 ():+: a unfoldr:: (s \rightarrow ():+: (a,s)) \rightarrow s \rightarrow [a]

Representing strings as streams

Goal

- ▶ Abstract from the concrete representation.
- Allow different access patterns.

Representing strings as streams

Goal

- ▶ Abstract from the concrete representation.
- Allow different access patterns.
- ▶ Idea: Use the unfoldr as representation.

Representing strings as streams

Goal

- ▶ Abstract from the concrete representation.
- Allow different access patterns.
- ▶ Idea: Use the unfoldr as representation.

```
\begin{split} & \text{unfoldr} :: (s \to \mathsf{Maybe}\ (a,s)) \to s \to [a] \\ & \text{unfoldr}\ \mathsf{next}\ s = \\ & \textbf{case}\ \mathsf{next}\ s\ \textbf{of} \\ & \mathsf{Nothing}\ \to [] \\ & \mathsf{Just}\ (x,r) \to x : \mathsf{unfoldr}\ \mathsf{next}\ r \end{split}
```

data Stream $s = Stream (s \rightarrow Maybe (Word8, s)) s$



Representing strings as streams (contd.)

 $\textbf{data} \ \mathsf{Stream} \ \mathsf{s} = \mathsf{Stream} \ (\mathsf{s} \to \mathsf{Maybe} \ (\mathsf{Word8}, \mathsf{s})) \ \mathsf{s}$

Problems

- ▶ We are not interested in the type s, we only care that we can apply the function to the seed.
- ► For efficiency reasons, it is good to have the length of the string.
- ▶ Also for efficiency reasons, it turns out to be good to have a third option next to "end of stream" and "next character": an explicit delay.

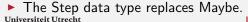
Representing strings as streams (contd.)

```
 \begin{tabular}{ll} \textbf{data} \ Stream &= \forall s.Stream \ (s \rightarrow Step \ s) \ s \ Int \\ \textbf{data} \ Step \ s &= Done \\ & | \ Yield \ Word8 \ s \\ & | \ Skip \ s \\ \end{tabular}
```

- Stream is a so-called existential type.
- Some people think ∀ should be ∃, but both views are valid ("forall s, there is a constructor such that ..." vs. "if you destruct a stream, there exists an s such that ...").
- ▶ The type of the constructor is

$$\mathsf{Stream} :: \forall \mathsf{s}. (\mathsf{s} \to \mathsf{Step} \; \mathsf{s}) \to \mathsf{s} \to \mathsf{Stream}$$

The s does not occur in the result type.







Building a stream

```
\begin{split} \text{readUp} &:: \mathsf{ByteString} \to \mathsf{Stream} \\ \text{readUp} &\: \mathsf{s} = \mathsf{Stream} \; \mathsf{next} \; 0 \; \mathsf{n} \\ &\: &\: \mathsf{where} \\ &\: \mathsf{n} &\: = \mathsf{length} \; \mathsf{s} \\ &\: \mathsf{next} \; \mathsf{i} \; | \; \mathsf{i} < \mathsf{n} &\: = \mathsf{Yield} \; (\mathsf{s} \, ! \, \mathsf{i}) \; (\mathsf{i} + 1) \\ &\: | \; \mathsf{otherwise} = \mathsf{Done} \end{split}
```

- ▶ We assume an array interface to ByteString internally.
- ▶ Other access patterns can easily be implemented.





Writing a stream

```
\label{eq:writeUp} \begin{split} \text{writeUp} :: \mathsf{Stream} &\to \mathsf{ByteString} \\ \text{writeUp} \ (\mathsf{Stream} \ \mathsf{next} \ \mathsf{s} \ \mathsf{n}) = \mathsf{listArray} \ (0, \mathsf{n} - 1) \\ & (\mathsf{unfoldStream} \ \mathsf{next} \ \mathsf{s}) \\ \hline \\ \textbf{where} \\ & \mathsf{unfoldStream} \ \mathsf{next} \ \mathsf{s} = \\ & \textbf{case} \ \mathsf{next} \ \mathsf{s} \ \textbf{of} \\ & \mathsf{Done} \quad \to [] \\ & \mathsf{Yield} \ \mathsf{x} \ \mathsf{r} \to \mathsf{x} : \mathsf{unfoldStream} \ \mathsf{next} \ \mathsf{r} \\ & \mathsf{Skip} \ \mathsf{r} \quad \to \quad \mathsf{unfoldStream} \ \mathsf{next} \ \mathsf{r} \end{split}
```



Modifying a stream

```
\begin{split} \text{map} &:: (\mathsf{Word8} \to \mathsf{Word8}) \to \mathsf{ByteString} \to \mathsf{ByteString} \\ \text{map} & f = \mathsf{writeUp} \circ \mathsf{mapS} \ \mathsf{f} \circ \mathsf{readUp} \\ \text{mapS} &:: (\mathsf{Word8} \to \mathsf{Word8}) \to \mathsf{Stream} \to \mathsf{Stream} \\ \text{mapS} & f \ (\mathsf{Stream} \ \mathsf{next} \ \mathsf{s} \ \mathsf{n}) = \mathsf{Stream} \ \mathsf{next}' \ \mathsf{s} \ \mathsf{n} \\ \textbf{where} \\ \text{next}' & \mathsf{s} = \textbf{case} \ \mathsf{next} \ \mathsf{s} \ \textbf{of} \\ \text{Done} \\ \text{Yield} & \mathsf{x} \ \mathsf{r} \to \mathsf{Yield} \ (\mathsf{f} \ \mathsf{x}) \ \mathsf{r} \\ \text{Skip} & \mathsf{r} \to \mathsf{Skip} \ \mathsf{r} \end{split}
```

Note that mapS is not recursive.

Stream fusion

```
\begin{array}{l} \text{map } f \circ \text{map } g \\ \equiv \quad \big\{ \begin{array}{l} \text{Definition of map, twice} \ \big\} \\ \text{writeUp} \circ \text{mapS } f \circ \text{readUp} \circ \text{writeUp} \circ \text{mapS } g \circ \text{readUp} \\ \equiv \quad \big\{ \begin{array}{l} \text{readUp/writeUp fusion via GHC rewrite rule} \ \big\} \\ \text{writeUp} \circ \text{mapS } f \circ \text{mapS } g \circ \text{readUp} \\ \equiv \quad \big\{ \begin{array}{l} \text{GHC unfolding of non-recursive functions} \ \big\} \\ \text{writeUp} \circ \text{mapS} \left( f \circ g \right) \circ \text{readUp} \end{array} \end{array}
```

4日 > 4 個 > 4 豆 > 4 豆 > 豆 めの()

GHC rewrite rules

- GHC has a scriptable optimizer.
- Rewrite rules such as

```
\mathsf{readUp} \circ \mathsf{writeUp} = \mathsf{id}
```

can be passed to GHC in pragmas.

GHC syntax:

```
{-# RULES
"readUp/writeUp"
forall x. (readUp (writeUp x)) = x
#-}
```

► The rules are type checked, but the user is responsible for their correctness!



Summary

- ▶ Lists are suitable only for a limited number of operations.
- Standard immutable arrays are only an option if updates are rare.
- ► Imperative arrays or Diff arrays are good option if fast access is desired and persistence is not required.
- ▶ ByteStrings are a fast and compact alternative for the Haskell String type. Use them for processing large strings (or files).

Next lecture

- ▶ for Tuesday read the "Finger Trees" paper (see wiki)
- ► For next week, read "Tackling the Awkward Squad" (see Wiki).
- ▶ New set of weekly assignments today/tomorrow.