Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Advanced Functional Programming

2012-2013, periode 2

Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

Jan 8, 2012

14. 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 . . .

14.1 Lists everywhere

What are lists good for?

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

What are lists good for?

$$
\begin{aligned}
&\mathsf{head} :: [a] \to a \\
&\mathsf{tail} \quad :: [a] \to [a] \\
&(:) \quad\; :: a \to [a] \to [a]
\end{aligned}
$$

- These are efficient operations on lists.

What are lists good for?

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

- These are efficient operations on lists.

What are lists good for?

```
top  :: [a] → a              -- O(1)
pop  :: [a] → [a]            -- O(1)
push :: a → [a] → [a]        -- O(1)
```

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

What are lists good for?

```
top  :: [a] → a           -- O(1)
pop  :: [a] → [a]         -- O(1)
push :: a → [a] → [a]     -- O(1)
```

- 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

```
snoc    :: [a] → a → [a]                 -- O(n)
snoc    = λxs x → xs ++ [x]
(!!)    :: [a] → Int → a                 -- O(n)
(++)    :: [a] → [a] → [a]               -- O(n)
reverse :: [a] → [a]                     -- O(n), naively: O(n^2)
union   :: Eq a ⇒ [a] → [a] → [a]        -- O(mn)
elem    :: Eq a ⇒ a → [a] → Bool         -- O(n)
```


Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Other operations on lists

```
snoc    :: [a] → a → [a]                   -- O(n)
snoc    = λxs x → xs ++ [x]
(!!)    :: [a] → Int → a                   -- O(n)
(++)    :: [a] → [a] → [a]                 -- O(n)
reverse :: [a] → [a]                       -- O(n), naively: O(n^2)
union   :: Eq a ⇒ [a] → [a] → [a]          -- O(mn)
elem    :: Eq a ⇒ a → [a] → Bool           -- O(n)
```

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.

14.2 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     :: (Ix i) ⇒ (i, i) → [(i, e)] → Array i e
listArray :: (Ix i) ⇒ (i, i) → [e] → Array i e

-- lookup
(!)       :: (Ix i) ⇒ Array i e → i → e
bounds    :: (Ix i) ⇒ Array i e → (i, i)

-- update
(//)      :: (Ix i) ⇒ Array i e → [(i, e)] → Array i e

-- destruction
elems     :: (Ix i) ⇒ Array i e → [e]
assocs    :: (Ix i) ⇒ Array i e → [(i, e)]
```

Interface of mutable arrays

Data.Array.IO

```
data IOArray i e    -- abstract
-- creation
newArray     :: (Ix i) ⇒ (i, i) → e → IO (IOArray i e)
newListArray :: (Ix i) ⇒ (i, i) → [e] → IO (IOArray i e)
-- lookup
readArray    :: (Ix i) ⇒ IOArray i e → i → IO e
getBounds    :: (Ix i) ⇒ IOArray i e → IO (i, i)
-- update
writeArray   :: (Ix i) ⇒ IOArray i e → i → e → IO ()
-- destruction
getElems     :: (Ix i) ⇒ Array i e → IO [e]
getAssocs    :: (Ix i) ⇒ Array i e → IO [(i, e)]
```

Conversion

$$\begin{aligned}
&\mathsf{freeze} :: (\mathsf{Ix}\ \mathsf{i}) \Rightarrow \mathsf{IOArray}\ \mathsf{i}\ \mathsf{e} \rightarrow \mathsf{IO}\ (\mathsf{Array}\ \mathsf{i}\ \mathsf{e})\\
&\mathsf{thaw}\ \ :: (\mathsf{Ix}\ \mathsf{i}) \Rightarrow \mathsf{Array}\ \mathsf{i}\ \mathsf{e} \rightarrow \mathsf{IO}\ (\mathsf{IOArray}\ \mathsf{i}\ \mathsf{e})
\end{aligned}$$

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 $O(1)$.

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.

14.3 Unboxed types

Unboxed types

```
Prelude⟩ :i Char
data Char = GHC.Base.C# GHC.Prim.Char#
```

- 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.

14.4 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:

$$\begin{aligned}&\mathsf{return} \circ \mathsf{foldl'}\ \mathsf{hash}\ 5381 \circ \mathsf{map}\ \mathsf{toLower} \circ\\ &\qquad\quad \mathsf{filter}\ \mathsf{isAlpha} \mathrel{=\!\!<\!\!<} \mathsf{readFile}\ \mathsf{f}\\ &\quad \textbf{where}\ \mathsf{hash}\ \mathsf{h}\ \mathsf{c} = \mathsf{h} * 33 + \mathsf{ord}\ \mathsf{c}\end{aligned}$$

$$\begin{aligned}&(\mathrel{=\!\!<\!\!<}) :: (\mathsf{a} \to \mathsf{IO}\ \mathsf{b}) \to \mathsf{IO}\ \mathsf{a} \to \mathsf{IO}\ \mathsf{b}\\ &(\mathrel{>\!\!>\!\!=}) :: \mathsf{IO}\ \mathsf{a} \to (\mathsf{a} \to \mathsf{IO}\ \mathsf{b}) \to \mathsf{IO}\ \mathsf{b}\end{aligned}$$

- 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:

$$\begin{aligned}&\mathsf{return} \circ \mathsf{foldl'}\ \mathsf{hash}\ 5381 \circ \mathsf{map}\ \mathsf{toLower} \circ \\ &\qquad\quad \mathsf{filter}\ \mathsf{isAlpha} \mathrel{=\!\!\ll} \mathsf{readFile}\ \mathsf{f} \\ &\quad \textbf{where}\ \mathsf{hash}\ \mathsf{h}\ \mathsf{c} = \mathsf{h} * 33 + \mathsf{ord}\ \mathsf{c}\end{aligned}$$

$$\begin{aligned}&(\mathrel{=\!\!\ll}) :: (\mathsf{a} \to \mathsf{IO}\ \mathsf{b}) \to \mathsf{IO}\ \mathsf{a} \to \mathsf{IO}\ \mathsf{b} \\ &(\gg\!\!=) :: \mathsf{IO}\ \mathsf{a} \to (\mathsf{a} \to \mathsf{IO}\ \mathsf{b}) \to \mathsf{IO}\ \mathsf{b}\end{aligned}$$

- 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.

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 corresponding callsparameters of the fold."

Excursion: unfoldr

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

```
unfoldr :: (s → Maybe (a, s)) → s → [a]
unfoldr next s =
  case next s of
    Nothing   → []
    Just (x, r) → x : unfoldr next r
```

Excursion: unfoldr

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

```
unfoldr :: (s → Maybe (a, s)) → s → [a]
unfoldr next s =
  case next s of
    Nothing    → []
    Just (x, r) → x : unfoldr next r
```

```
repeat          = unfoldr (λx → Just (x, x))
replicate n x   = unfoldr (λn → if n == 0 then Nothing
                                  else Just (x, n − 1)) n
enumFromTo b e  = unfoldr (λb → if b > e  then Nothing
                                  else Just (b, b + 1)) b
```

unfoldr vs. foldr

$$\begin{aligned}
\mathsf{unfoldr} &:: (\mathsf{s} \rightarrow \mathsf{Maybe}\ (\mathsf{a}, \mathsf{s})) \rightarrow \mathsf{s} \rightarrow [\mathsf{a}] \\
\mathsf{foldr} &:: (\mathsf{a} \rightarrow \mathsf{r} \rightarrow \mathsf{r}) \rightarrow \mathsf{r} \rightarrow [\mathsf{a}] \rightarrow \mathsf{r}
\end{aligned}$$

unfoldr vs. foldr

$$\text{unfoldr} :: (s \to \text{Maybe}\ (a, s)) \to s \to [a]$$
$$\text{foldr} \quad :: (a \to r \to r) \to r \to [a] \to r$$

$$\text{foldr} :: (r, a \to r \to r) \to [a] \to r$$
$$\text{foldr} :: (() : + : (a, r) \to r) \to [a] \to r$$

unfoldr vs. foldr

$$\text{unfoldr} :: (s \to \text{Maybe}\ (a, s)) \to s \to [a]$$
$$\text{foldr}\ \ :: (a \to r \to r) \to r \to [a] \to r$$

$$\text{foldr} :: (r, a \to r \to r) \to [a] \to r$$
$$\text{foldr} :: (() :+: (a, r) \to r) \to [a] \to r$$

$$\text{Maybe}\ a \approx () :+: a$$
$$\text{unfoldr} :: (s \to () :+: (a, s)) \to s \to [a]$$

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.

```
unfoldr :: (s → Maybe (a, s)) → s → [a]
unfoldr next s =
  case next s of
    Nothing   → [ ]
    Just (x, r) → x : unfoldr next r
```

```
data Stream s = Stream (s → Maybe (Word8, s)) s
```

Representing strings as streams (contd.)

$$\textbf{data}\ \text{Stream s} = \text{Stream}\ (s \rightarrow \text{Maybe}\ (\text{Word8}, s))\ s$$

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.)

$$\textbf{data}\ \text{Stream} = \forall s.\text{Stream}\ (s \to \text{Step}\ s)\ s\ \text{Int}$$

$$\begin{aligned}\textbf{data}\ \text{Step}\ s = {} & \text{Done} \\ \mid {} & \text{Yield Word8}\ s \\ \mid {} & \text{Skip}\ s\end{aligned}$$

- 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

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

  The s does not occur in the result type.
- The Step data type replaces Maybe.

Building a stream

$$
\begin{aligned}
&\mathsf{readUp} :: \mathsf{ByteString} \rightarrow \mathsf{Stream} \\
&\mathsf{readUp}\ \mathsf{s} = \mathsf{Stream}\ \mathsf{next}\ 0\ \mathsf{n} \\
&\quad \textbf{where} \\
&\qquad \mathsf{n} \qquad\qquad\qquad\quad = \mathsf{length}\ \mathsf{s} \\
&\qquad \mathsf{next}\ \mathsf{i} \mid \mathsf{i} < \mathsf{n} \qquad = \mathsf{Yield}\ (\mathsf{s}\,!\,\mathsf{i})\ (\mathsf{i} + 1) \\
&\qquad\qquad\ \ \mid \mathsf{otherwise} = \mathsf{Done}
\end{aligned}
$$

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

Writing a stream

```
writeUp :: Stream → ByteString
writeUp (Stream next s n) = listArray (0, n − 1)
                                      (unfoldStream next s)
  where
    unfoldStream next s =
      case next s of
        Done     → []
        Yield x r → x : unfoldStream next r
        Skip r   →     unfoldStream next r
```

Modifying a stream

```
map :: (Word8 → Word8) → ByteString → ByteString
map f = writeUp ∘ mapS f ∘ readUp

mapS :: (Word8 → Word8) → Stream → Stream
mapS f (Stream next s n) = Stream next' s n
  where
    next' s = case next s of
                Done
                Yield x r → Yield (f x) r
                Skip r    → Skip r
```

Note that mapS is not recursive.

Stream fusion

$$
\begin{aligned}
& \text{map f} \circ \text{map g} \\
\equiv \quad & \{ \text{ Definition of map, twice } \} \\
& \text{writeUp} \circ \text{mapS f} \circ \text{readUp} \circ \text{writeUp} \circ \text{mapS g} \circ \text{readUp} \\
\equiv \quad & \{ \text{ readUp/writeUp fusion via GHC rewrite rule } \} \\
& \text{writeUp} \circ \text{mapS f} \circ \text{mapS g} \circ \text{readUp} \\
\equiv \quad & \{ \text{ GHC unfolding of non-recursive functions } \} \\
& \text{writeUp} \circ \text{mapS (f} \circ \text{g)} \circ \text{readUp}
\end{aligned}
$$

GHC rewrite rules

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

  > $\mathrm{readUp} \circ \mathrm{writeUp} = \mathrm{id}$

  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).