Data Structures

The Practice of Haskell Programming

Andres Löh

Well-Typed

May 18, 2012

Overview

- Persistent data structures
- Arrays
- Trees, sets and finite maps
- Other useful data structures

Persistent data structures

Imperative vs. functional style

Given a finite map (associative map, dictionary) `m`.

Imperative style

```
foo.put (42, "Bar"); ...
```

Functional style

```
let foo′ = insert 42 "Bar" m in ...
```

What is the difference?

Imperative vs. functional style

Given a finite map (associative map, dictionary) `m`.

Imperative style

```
foo.put (42, "Bar"); ...
```

Functional style

```
let foo' = insert 42 "Bar" m in ...
```

What is the difference?

Imperative: destructive update

Functional: creation of a new value

Persistent data structures

Imperative languages:

- many operations make use of destructive updates
- after an update, the old version of the data structure is no longer available

Persistent data structures

Imperative languages:

- many operations make use of destructive updates
- after an update, the old version of the data structure is no longer available

Functional languages:

- most operations create a new data structure
- old versions are still available

Persistent data structures

Imperative languages:

- many operations make use of destructive updates
- after an update, the old version of the data structure is no longer available

Functional languages:

- most operations create a new data structure
- old versions are still available

Data structures where old version remain accessible are called **persistent**.

Persistent data structures

- In functional languages, most data structures are (automatically) persistent.
- In imperative languages, most data structures are not persistent (**ephemeral**).
- It is generally possible to also use ephemeral data structures in functional or persistent data structures in imperative languages.

Persistent data structures

- In functional languages, most data structures are (automatically) persistent.
- In imperative languages, most data structures are not persistent (**ephemeral**).
- It is generally possible to also use ephemeral data structures in functional or persistent data structures in imperative languages.

How do persistent data structures work?

Example: Haskell lists

$[1,2,3,4]$

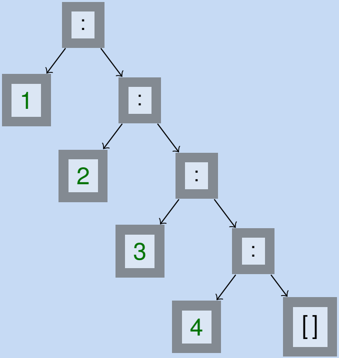
Example: Haskell lists

$[1, 2, 3, 4]$ is syntactic sugar for $1 : (2 : (3 : (4 : [\,])))$

Example: Haskell lists

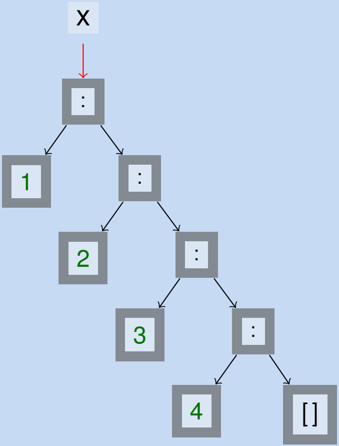
$[1, 2, 3, 4]$ is syntactic sugar for $1 : (2 : (3 : (4 : [\,])))$

Representation in memory:



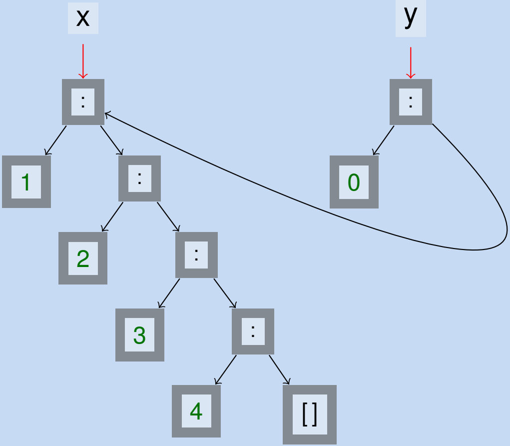
Lists are persistent

$\mathbf{let}\ x = [1, 2, 3, 4]; y = 0 : x; z = \mathrm{drop}\ 2\ y\ \mathbf{in} \ldots$



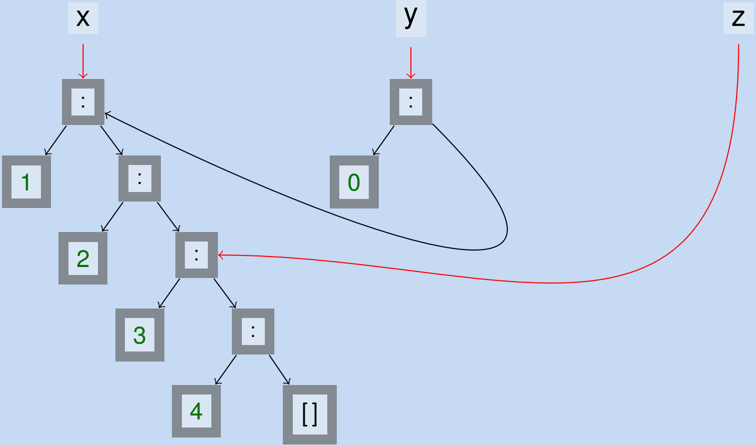
Lists are persistent

$\mathbf{let}\ x = [1, 2, 3, 4]; y = 0 : x; z = \mathrm{drop}\ 2\ y\ \mathbf{in} \ldots$



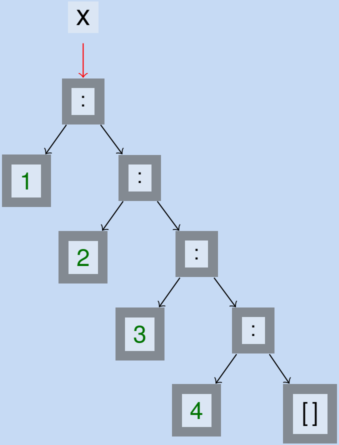
Lists are persistent

$\textbf{let}\ x = [1,2,3,4]; y = 0 : x; z = \text{drop}\ 2\ y\ \textbf{in} \ldots$



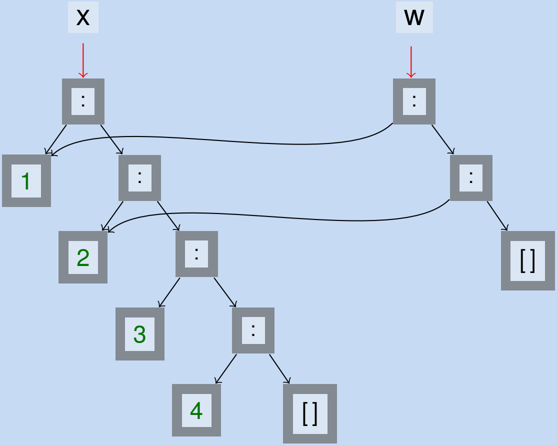
Lists are persistent – contd.

let x = [1, 2, 3, 4]; w = take 2 x **in** . . .



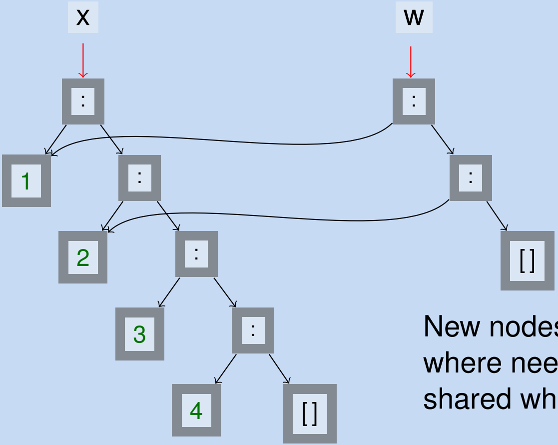
Lists are persistent – contd.

$\textbf{let}\ x = [1, 2, 3, 4]; w = \text{take}\ 2\ x\ \textbf{in} \ldots$



Lists are persistent – contd.

$\textbf{let}\ x = [1, 2, 3, 4]; w = \text{take}\ 2\ x\ \textbf{in} \ldots$



New nodes are allocated where needed; nodes are shared where possible.

Implementation of persistent data structures

- Modifications of an existing structure take place by creating new nodes and pointers.
- Sometimes, parts of a structure have to be copied, because the old version must not be modified.

Of course, we want to copy as little as possible, and reuse as much as possible.

Vacuum

Vacuum

Vacuum is a library originally developed by Matt Morrow:

- the library is a debugging tool,
- we can query and generate the internal graph representation of Haskell terms,
- useful to understand how Haskell terms are shared.

There are several visualization layers for vacuum available from Hackage. Unfortunately, many of them are somewhat tricky to build.

Vacuum

Vacuum is a library originally developed by Matt Morrow:

- the library is a debugging tool,
- we can query and generate the internal graph representation of Haskell terms,
- useful to understand how Haskell terms are shared.

There are several visualization layers for vacuum available from Hackage. Unfortunately, many of them are somewhat tricky to build.

```
〉 view (let x = [1, 2] in x ++ x)
〉 view (let x = [1, 2, 3, 4] y = 0 : x; z = drop 3 y in (x, y, z))
〉 view (let x = [1, 2, 3, 4]; w = take 2 x in (x, w))
〉 view (repeat 1)
```

Arrays

Persistence and complexity

Some data structures show unexpected (i.e., bad) behaviour when used in a persistent setting:

Persistence and complexity

Some data structures show unexpected (i.e., bad) behaviour when used in a persistent setting:

Haskell arrays:

$$\textbf{let}\ x = \text{listArray}\ (0, 4)\ [1, 2, 3, 4, 5]\ \textbf{in}\ x\ //\ [(2, 13)]$$

Persistence and complexity

Some data structures show unexpected (i.e., bad) behaviour when used in a persistent setting:

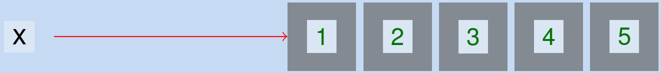
Haskell arrays:

$\mathbf{let}\ x = \mathrm{listArray}\ (0, 4)\ [1, 2, 3, 4, 5]\ \mathbf{in}\ x\ //\ [(2, 13)]$

How expensive is the update operation?

- In an imperative language, we expect O(1), i.e., constant time.

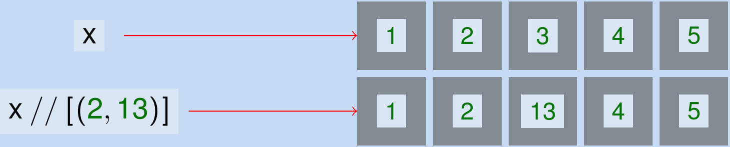
Arrays




```
x // [(2, 13)]
```

- Arrays are stored in a contiguous block of memory.
- This allows O(1) access to each element.
- In an imperative setting, a destructive update is also possible in O(1).

Arrays



- Arrays are stored in a contiguous block of memory.
- This allows O(1) access to each element.
- In an imperative setting, a destructive update is also possible in O(1).
- But if a persistent update is desired, the whole array must be copied, which takes O(n), i.e., linear time.

Advice on arrays

Be careful when using them:

- stay away if you require a large number of incremental updates – finite maps are usually much better then;
- arrays can be useful if you have an essentially constant table that you need to access frequently;
- arrays can also be useful if you perform global updates on them anyway.

The `vector` package

There's a new, quite popular array package available from Hackage called `vector`:

- Developed by Roman Leshchinskiy.
- An interface capturing mutable and immutable arrays, boxed and unboxed arrays in a slightly more systematic way than the standard Haskell array interface allows.
- Support slicing operations.

Trees

Trees

- Arrays and hash tables are expensive in a functional (persistent) setting, because it is impossible to share substructures between different versions.
- Tree-shaped structures, however, are generally very suitable in a functional setting. Reuse of subtrees is easy to achieve. Most functional data structures therefore are some sort of trees.

Trees

- Arrays and hash tables are expensive in a functional (persistent) setting, because it is impossible to share substructures between different versions.
- Tree-shaped structures, however, are generally very suitable in a functional setting. Reuse of subtrees is easy to achieve. Most functional data structures therefore are some sort of trees.

Lists are trees, too – just a very peculiar variant.

Lists

- There is a lot of syntactic sugar for lists in Haskell. Thus, lists are used for a lot of different purposes.
- Lists are the default data structure in functional languages much as arrays are in imperative languages.
- However, lists support only **very few operations efficiently**.

Operations on lists

```
[]      :: [a]                                   -- O(1)
(:)     :: a → [a] → [a]                         -- O(1)
head    :: [a] → a                               -- O(1)
tail    :: [a] → [a]                             -- O(1)
snoc    :: [a] → a → [a]                         -- O(n)
snoc    = λxs x → xs ++ [x]
(!!)    :: [a] → Int → a                         -- O(n)
(++)    :: [a] → [a] → [a]                       -- O(m), first list
reverse :: [a] → [a]                             -- O(n)
splitAt :: Int → [a] → ([a], [a])                -- O(n)
union   :: Eq a ⇒ [a] → [a] → [a]                -- O(mn)
elem    :: Eq a ⇒ a → [a] → Bool                 -- O(n)
```

Guidelines for using lists

Lists are suitable for use if:

- most operations we need are **stack operations**,
- or the maximal size of the lists we deal with is relatively small,

A special case of stack-like access is if we traverse a large list linearly.

Guidelines for using lists

Lists are suitable for use if:

- most operations we need are **stack operations**,
- or the maximal size of the lists we deal with is relatively small,

A special case of stack-like access is if we traverse a large list linearly.

Lists are generally not suitable:

- for random access,
- for set operations such as union and intersection,
- to deal with (really) large amounts of texts as `String`.

What is better than lists?

Are there functional data structures that support a more efficient lookup operation than lists?

What is better than lists?

Are there functional data structures that support a more efficient lookup operation than lists?

Yes, balanced search trees.

What is better than lists?

Are there functional data structures that support a more efficient lookup operation than lists?

Yes, balanced search trees.

Can be used to implement finite maps and sets efficiently, and persistently.

What is better than lists?

Are there functional data structures that support a more efficient lookup operation than lists?

Yes, balanced search trees.

Can be used to implement finite maps and sets efficiently, and persistently.

Question

What is a (binary) search tree?

Finite maps

- A finite map is a function with a finite domain (type of **keys**).
- Useful for a wide variety of applications (tables, environments, “arrays”).
- Inefficient representation: **type** Map a b $=$ $[(a, b)]$.

An efficient implementation of finite maps

- Based on binary search trees.
- Available in `Data.Map` and `Data.IntMap` for `Int` as key type.
- Provided by the `containers` package that is part of the Haskell Platform.
- Keys are stored ordered in the tree, so that efficient lookup is possible.
- Requires the keys to be ordered.
- Inserting and removing elements can trigger rotations to rebalance the tree.
- Everything happens in a persistent setting.

Sets

- Sets are a special case of finite maps: essentially,

```
type Set a = Map a ()
```

- A specialized set implementation is available in `Data.Set` and `Data.IntSet`, but the idea is the same as for finite maps.

Finite map interface

This is an excerpt from the functions available in `Data.Map`:

```
data Map k a          -- abstract

insert :: (Ord k) ⇒ k → a → Map k a → Map k a        -- O(log n)
lookup :: (Ord k) ⇒ k → Map k a → Maybe a            -- O(log n)
delete :: (Ord k) ⇒ k → Map k a → Map k a            -- O(log n)
update :: (Ord k) ⇒ (a → Maybe a) →
                    k → Map k a → Map k a            -- O(log n)
union  :: (Ord k) ⇒ Map k a → Map k a → Map k a      -- O(m + n)
member :: (Ord k) ⇒ k → Map k a → Bool               -- O(log n)
size   :: Map k a → Int                              -- O(1)
map    :: (a → b) → Map k a → Map k b                -- O(n)
```

The interface for `Set` is very similar.

Implementing finite maps

Implementation

In the following, we will sketch the implementation as it is available in `Data.Map`:

```haskell
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

Implementation

In the following, we will sketch the implementation as it is available in `Data.Map`:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The `!` is a strictness annotation for extra efficiency. More about that later.

Implementation

In the following, we will sketch the implementation as it is available in `Data.Map`:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)

type Size = Int
```

The `!` is a strictness annotation for extra efficiency. More about that later.

A map is

- either a leaf called `Tip`,

Implementation

In the following, we will sketch the implementation as it is available in `Data.Map`:

```haskell
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The `!` is a strictness annotation for extra efficiency. More about that later.

A map is

- either a leaf called `Tip`,
- or a binary node called `Bin`

Implementation

In the following, we will sketch the implementation as it is available in `Data.Map`:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The `!` is a strictness annotation for extra efficiency. More about that later.

A map is

- either a leaf called `Tip`,
- or a binary node called `Bin` containing
  - the size of the tree,

Implementation

In the following, we will sketch the implementation as it is available in `Data.Map`:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The `!` is a strictness annotation for extra efficiency. More about that later.

A map is

- either a leaf called `Tip`,
- or a binary node called `Bin` containing
  - the size of the tree,
  - the key value pair,

Implementation

In the following, we will sketch the implementation as it is available in `Data.Map`:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The `!` is a strictness annotation for extra efficiency. More about that later.

A map is

- either a leaf called `Tip`,
- or a binary node called `Bin` containing
  - the size of the tree,
  - the key value pair,
  - and a left and right subtree.

Creating finite maps

```
empty :: Map k a
empty = Tip

singleton :: k → a → Map k a
singleton k x = bin Tip k x Tip
```

Creating finite maps

```
empty :: Map k a
empty = Tip

singleton :: k → a → Map k a
singleton k x = bin Tip k x Tip
```

The function `bin` is an example of a **smart constructor** ...

Smart constructors

Smart constructors are wrappers around constructors that help to ensure that invariants of the data structure are maintained.

Smart constructors

Smart constructors are wrappers around constructors that help to ensure that invariants of the data structure are maintained.

In this case: the Size argument of Bin should always reflect the actual size of the tree.

```
bin :: Map k a → k → a → Map k a → Map k a
bin l kx x r = Bin (size l + size r + 1) l kx x r
```

Smart constructors

Smart constructors are wrappers around constructors that help to ensure that invariants of the data structure are maintained.

In this case: the `Size` argument of `Bin` should always reflect the actual size of the tree.

```
bin :: Map k a → k → a → Map k a → Map k a
bin l kx x r = Bin (size l + size r + 1) l kx x r
```

```
size :: Map k a → Int
size Tip                = 0
size (Bin sz _ _ _ _) = sz
```

Finding an element

```
lookup :: Ord k ⇒ k → Map k a → Maybe a
lookup key Tip              = Nothing
lookup key (Bin _ l kx x r) =
  case compare key kx of
    LT → lookup key l
    GT → lookup key r
    EQ → Just x
```

Finding an element

```
lookup :: Ord k ⇒ k → Map k a → Maybe a
lookup key Tip                = Nothing
lookup key (Bin _ l kx x r) =
   case compare key kx of
      LT  → lookup key l
      GT → lookup key r
      EQ → Just x
```

Comparing two elements:

```
compare :: Ord a ⇒ a → a → Ordering
data Ordering = LT | EQ | GT
```

Inserting an element

```
insert :: Ord k ⇒ k → a → Map k a → Map k a
insert kx x Tip              = singleton kx x    -- insert new
insert kx x (Bin sz l ky y r) =
  case compare kx ky of
    LT → balance (insert kx x l) ky y              r
    GT → balance                 l  ky y (insert kx x r)
    EQ → Bin sz l kx x r    -- replace old
```

Inserting an element

```
insert :: Ord k ⇒ k → a → Map k a → Map k a
insert kx x Tip              = singleton kx x    -- insert new
insert kx x (Bin sz l ky y r) =
  case compare kx ky of
    LT → balance (insert kx x l) ky y              r
    GT → balance                 l  ky y (insert kx x r)
    EQ → Bin sz l kx x r    -- replace old
```

The function `balance` is an even smarter constructor with the same type as `bin`:

```
balance :: Map k a → k → a → Map k a → Map k a
```

Balancing the tree

We could just define

$$\text{balance} = \text{bin}$$

and that would actually be correct.

Balancing the tree

We could just define

$\text{balance} = \text{bin}$

and that would actually be correct.

Question

What is the problem, and when does it arise?

Balancing approach

- If the height of the two subtrees is not too different, we just use `Bin`.
- Otherwise, we perform a rotation.

Balancing approach

- If the height of the two subtrees is not too different, we just use `Bin`.
- Otherwise, we perform a rotation.

Rotation

A rearrangement of the tree that preserves the search tree property.

Rotation

```
rotateL :: Map a b → a → b → Map a b → Map a b
rotateL l kx x r@(Bin _ ly _ _ ry)
    | size ly < ratio * size ry = singleL  l kx x r
    | otherwise                 = doubleL l kx x r
rotateL _ _ _ Tip = error "rotateL Tip"
```

Depending on the shape of the tree, either a simple (single) or a more complex (double) rotation is performed.

Rotation – contd.

```
singleL :: Map a b → a → b → Map a b → Map a b
singleL t1 k1 x1 (Bin _ t2 k2 x2 t3) =
  bin (bin t1 k1 x1 t2) k2 x2 t3
```

Rotation – contd.

```
singleL :: Map a b → a → b → Map a b → Map a b
singleL t1 k1 x1 (Bin _ t2 k2 x2 t3) =
  bin (bin t1 k1 x1 t2) k2 x2 t3
```

```
doubleL :: Map a b → a → b → Map a b → Map a b
doubleL t1 k1 x1 (Bin _ (Bin _ t2 k2 x2 t3) k3 x3 t4) =
  bin (bin t1 k1 x1 t2) k2 x2 (bin t3 k3 x3 t4)
```

Rotation – contd.

```
singleL :: Map a b → a → b → Map a b → Map a b
singleL t1 k1 x1 (Bin _ t2 k2 x2 t3) =
  bin (bin t1 k1 x1 t2) k2 x2 t3
```

```
doubleL :: Map a b → a → b → Map a b → Map a b
doubleL t1 k1 x1 (Bin _ (Bin _ t2 k2 x2 t3) k3 x3 t4) =
  bin (bin t1 k1 x1 t2) k2 x2 (bin t3 k3 x3 t4)
```

Note how easy it is to see that these rotations preserve the search tree property.

Other useful data structures

Finger trees

- A balanced persistent tree structure.
- Supports search tree operations in logarithmic time.
- Supports cons and snoc in $O(1)$.
- Supports logarithmic splitting and union.

Finger trees

- A balanced persistent tree structure.
- Supports search tree operations in logarithmic time.
- Supports cons and snoc in $O(1)$.
- Supports logarithmic splitting and union.

A universal data structure. A good choice for a wide range of applications.

Finger trees

- A balanced persistent tree structure.
- Supports search tree operations in logarithmic time.
- Supports cons and snoc in $O(1)$.
- Supports logarithmic splitting and union.

A universal data structure. A good choice for a wide range of applications.
Available in Data.Sequence from `containers` and in an extended version in the `fingertree` package on Hackage.

Byte strings and `Text`

Haskell strings are lists of characters.

Question

How much memory is needed to store a `String` that is three characters long?

Byte strings and `Text`

Haskell strings are lists of characters.

Question

How much memory is needed to store a `String` that is three characters long?

There are other suitable datatypes for strings:

- Byte strings are stored as compact arrays (provided by `bytestring`. Mainly suitable for low-level or binary data.
- `Text` (provided by `text`) is a convenient datatype for text that wraps `bytestring` and deals with encoding issues.
- To prevent the typical array problems, a clever form of optimization called stream fusion is being used.

More on Hackage

On Hackage, there are several additional libraries for data structures.

Some examples: heaps, priority search queues, hash maps, heterogeneous lists, zippers, tries, graphs, quadtrees, ...

Summary

- It is important to keep persistence in mind when thinking about functional data structures.
- Arrays should be used with care.
- Lists are ok for stack-like use or simple traversals.
- Good general-purpose data structures are sets, finite maps and sequences.