Intro, Datatype Internals and Data Structures

Advanced Haskell

Andres Löh

10–11 October 2013 —

Well-Typed
The Haskell Consultants

Advanced Haskell

Overview of the course

Today: Internals and Performance

We'll talk about data structures, Haskell's memory model, how to get information about the performance of Haskell programs, and about parallelism and concurrency.

Overview of the course

Today: Internals and Performance

We'll talk about data structures, Haskell's memory model, how to get information about the performance of Haskell programs, and about parallelism and concurrency.

Tomorrow: Patterns and Types

Topics include more advanced features of the type class system, type families, GADTs, how to express certain invariants with the type system, how to get the most out of monads and applicative functors.

Haskell and its extensions

The Haskell Standard

- The current version of the Haskell Standard is Haskell 2010.

The Haskell Standard

- The current version of the Haskell Standard is Haskell 2010.
- If you run GHC, you (more or less) get Haskell 2010 compliance.

The Haskell Standard

- The current version of the Haskell Standard is Haskell 2010.
- If you run GHC, you (more or less) get Haskell 2010 compliance.
- But GHC supports many language extensions, at various stages of maturity. Some of these will hopefully end up in future versions of the standard.

Listing and enabling language extensions

- Type `:set -X` and press TAB in GHCi to get a list of language extensions currently supported.
- You can enable language extensions by passing a `-X` flag for the extension in question to GHC or GHCi.
- The recommended way to enable language extensions is by using a LANGUAGE pragma at the top of your source file.

Compiler pragmas

A typical language pragma looks as follows:

```
{-# LANGUAGE GADTs, TypeFamilies #-}
```

- They look a bit like comments.
- The first word indicates the kind of pragma.
- A LANGUAGE pragma is followed by a comma-separated list of language extensions.
- LANGUAGE pragmas have to appear at the top of the file.
- There are several other kinds of pragmas that GHC understands.

In this course

In this course, we'll regularly go beyond what Haskell 2010 offers:

- I'll list the LANGUAGE pragmas needed in such cases.
- We only use extensions that are considered relatively stable; in some of the more controversial cases, I'll discuss it when we arrive at that point.

This morning

- We will start by looking at datatypes and data structures.
- We will also discuss how Haskell represents data in memory.
- As a preparation, we will learn a bit more about Haskell's type system.

Persistent data structures

Imperative vs. functional style

Given a finite map (associative map, dictionary) foo.

Imperative style

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

Functional style

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

What is the difference?

Imperative vs. functional style

Given a finite map (associative map, dictionary) foo.

Imperative style

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

Functional style

```
let foo′ = insert 42 "Bar" foo 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 (contd.)

- 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 (contd.)

- 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

$[1, 2, 3, 4]$

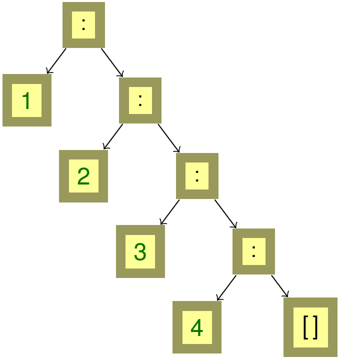
Example: Haskell lists

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

Example: Haskell lists

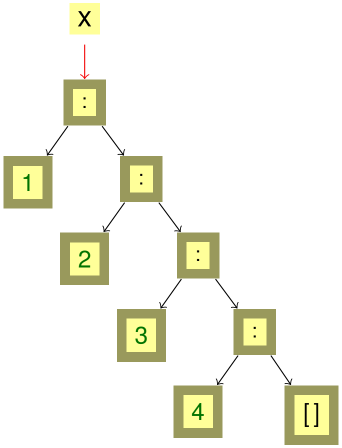
$[1,2,3,4]$ is syntactic sugar for $1 : (2 : (3 : (4 : [\,])))$

Representation in memory:



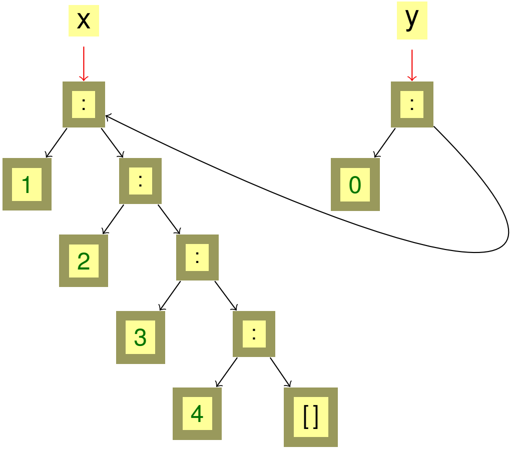
Lists are persistent

let x = [1, 2, 3, 4]; y = 0 : x; z = drop 3 y **in** . . .



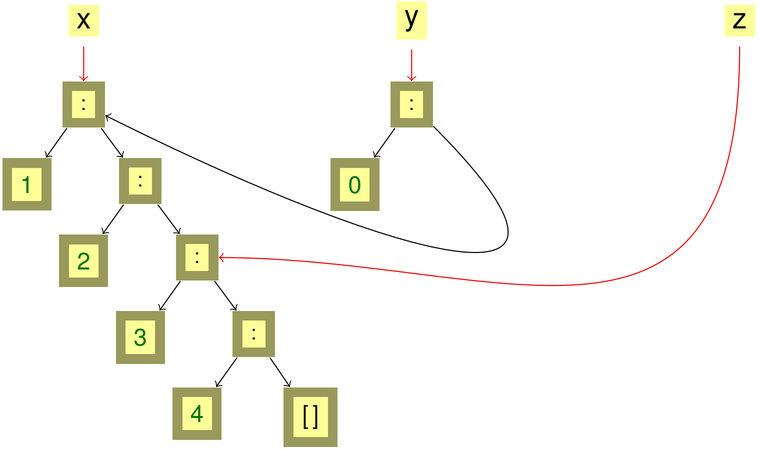
Lists are persistent

let x = [1, 2, 3, 4]; y = 0 : x; z = drop 3 y **in** . . .



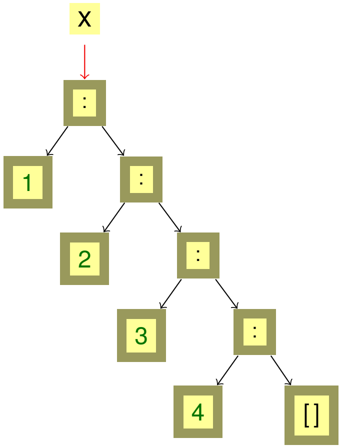
Lists are persistent

let x = [1, 2, 3, 4]; y = 0 : x; z = drop 3 y **in** . . .



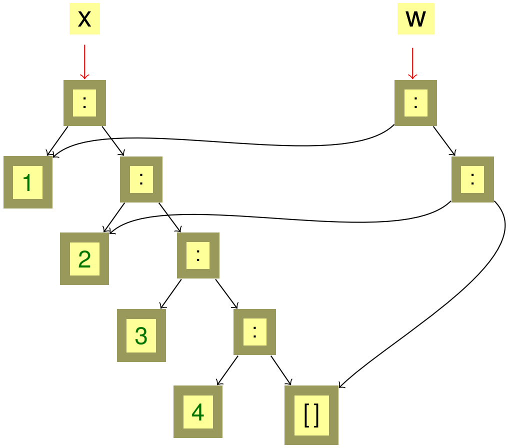
Lists are persistent (contd.)

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



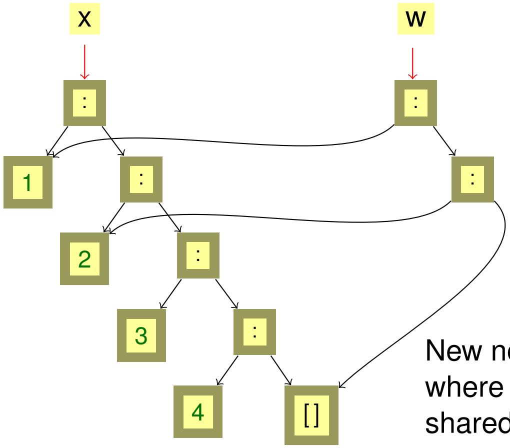
Lists are persistent (contd.)

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



Lists are persistent (contd.)

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



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.

Summary: representation of data on the heap

Values are represented using one or more words of memory:

- the first word is a tag that identifies the constructor;
- the other words are the payload, typically pointers to the arguments of the constructor.

Summary: representation of data on the heap

Values are represented using one or more words of memory:

- the first word is a tag that identifies the constructor;
- the other words are the payload, typically pointers to the arguments of the constructor.

Unevaluated data is represented using thunks:

- like a function, a thunk contains a code pointer that can be called to evaluate and update the thunk in-place.

Visualization tools

Visualizing the representation of data on the heap

`vacuum`

A library for inspecting the internal graph representation of Haskell terms, displaying sharing, but evaluating the inspected expression fully.

Several graphical frontends, but not all of them well-maintained and easy to install.

`ghc-vis` / `ghc-heap-view`

A library and graphical frontend similar to `vacuum`, but allows us to see unevaluated computations (thunk) and evaluate them interactively. Integration with GHCi.

ghc-vis example

Open the visualization window:

```
:vis
```

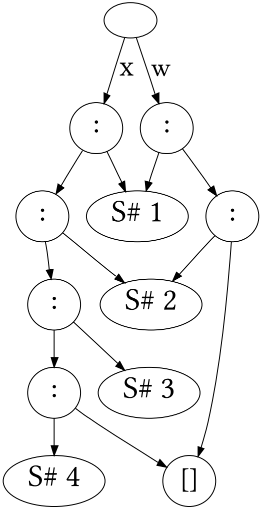
Then add terms to view; switch to graph view:

```
⟩ let x = [1, 2, 3, 4]; w = take 2 x
⟩ :view x
⟩ :view y
⟩ :switch
```

Evaluate `y` and update:

```
⟩ y
[1, 2]
⟩ :update
```

ghc-vis example (contd.)



Trees

Trees

Tree-shaped structures are generally very suitable for a persistent, functional setting:

- recursive structure of trees fits nicely with the natural way of writing recursive functions in Haskell;
- reuse / sharing of subtrees is easy to achieve, i.e., most operations on nodes just affect one path from the root to the node, and can reuse all other parts of the data structure.

Trees

Tree-shaped structures are generally very suitable for a persistent, functional setting:

- recursive structure of trees fits nicely with the natural way of writing recursive functions in Haskell;
- reuse / sharing of subtrees is easy to achieve, i.e., most operations on nodes just affect one path from the root to the node, and can reuse all other parts of the data structure.

Most functional data structures are some sort of trees.

Trees

Tree-shaped structures are generally very suitable for a persistent, functional setting:

- recursive structure of trees fits nicely with the natural way of writing recursive functions in Haskell;
- reuse / sharing of subtrees is easy to achieve, i.e., most operations on nodes just affect one path from the root to the node, and can reuse all other parts of the data structure.

Most functional data structures 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 text via `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.

Finite maps in the `containers` package

- A finite map is a function with a finite domain (type of keys).
- Useful for a wide variety of applications (tables, environments, “arrays”).
- Implementation based on binary search trees.
- Available in `Data.Map` and `Data.IntMap` for `Int` as key type.
- 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
empty  :: Map k a                                 -- O(1)
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.

A glimpse at the implementation

```
data Map k a = Tip
             | Bin {-# UNPACK #-} !Size
                   (Map k a) k a (Map k a)

type Size = Int
```

Well-Typed

A glimpse at the implementation

```
data Map k a = Tip
             | Bin {-# UNPACK #-} !Size
                   (Map k a) k a (Map k a)

type Size = Int
```

The `!` is a strictness annotation for extra efficiency. More about that later. Similarly the UNPACK pragma.

A glimpse at the implementation

```
data Map k a = Tip
             | Bin {-# UNPACK #-} !Size
                   (Map k a) k a (Map k a)
type Size = Int
```

The ! is a strictness annotation for extra efficiency. More about that later. Similarly the UNPACK pragma.

A map is

- either a leaf called Tip,

A glimpse at the implementation

```
data Map k a = Tip
             | Bin {-# UNPACK #-} !Size
                   (Map k a) k a (Map k a)

type Size = Int
```

The ! is a strictness annotation for extra efficiency. More about that later. Similarly the UNPACK pragma.

A map is

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

A glimpse at the implementation

```
data Map k a = Tip
             | Bin {-# UNPACK #-} !Size
                   (Map k a) k a (Map k a)

type Size = Int
```

The ! is a strictness annotation for extra efficiency. More about that later. Similarly the UNPACK pragma.

A map is

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

A glimpse at the implementation

```
data Map k a = Tip
             | Bin {-# UNPACK #-} !Size
                   (Map k a) k a (Map k a)

type Size = Int
```

The `!` is a strictness annotation for extra efficiency. More about that later. Similarly the UNPACK pragma.

A map is

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

A glimpse at the implementation

```
data Map k a = Tip
             | Bin {-# UNPACK #-} !Size
                   (Map k a) k a (Map k a)

type Size = Int
```

The ! is a strictness annotation for extra efficiency. More about that later. Similarly the UNPACK pragma.

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.

A glimpse at the implementation

```
data Map k a = Tip
             | Bin {-# UNPACK #-} !Size
                   (Map k a) k a (Map k a)

type Size = Int
```

The `!` is a strictness annotation for extra efficiency. More about that later. Similarly the UNPACK pragma.

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.

The important thing is that it's actually recognizable as a binary tree.

Smart constructors

The finite map library makes use of a common technique: smart constructors are wrappers around constructors that help to ensure that invariants of the data structure are maintained.

Smart constructors

The finite map library makes use of a common technique: 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

The finite map library makes use of a common technique: 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
```

Smart constructors

The finite map library makes use of a common technique: 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
```

If only bin rather than Bin is used to construct binary nodes, the size will always be correct.

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

```
balance = bin
```

and that would actually be correct.

Balancing the tree

We could just define

```
balance = bin
```

and that would actually be correct.

But certain sequences of `insert` would yield degenerated trees and make subsequent `lookup` calls quite costly.

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

Rotation – contd.

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

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

```
doubleL :: Map k a → k → a → Map k a → Map k a
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 – also, no pointer manipulations.

Sequences

Performance characteristics

Sometimes, we need a data structure with

- efficient random access to arbitrary elements;
- very efficient access to both ends;
- efficient concatenation and splitting.

Think of queues, pattern matching and extraction operations, search and replace operations, etc.

Performance characteristics

Sometimes, we need a data structure with

- efficient random access to arbitrary elements;
- very efficient access to both ends;
- efficient concatenation and splitting.

Think of queues, pattern matching and extraction operations, search and replace operations, etc.

This is offered by the Data.Sequence library, also from the `containers` package.

Sequence interface

Again, this is just a small excerpt:

```
data Seq a    -- abstract

empty :: Seq a                               -- O(1)
(◁)    :: a → Seq a → Seq a                   -- O(1)
(▷)    :: Seq a → a → Seq a                   -- O(1)
(×)    :: Seq a → Seq a → Seq a               -- O(log(min(m, n)))

null   :: Seq a → Bool                        -- O(1)
length :: Seq a → Int                         -- O(1)

filter :: (a → Bool) → Seq a → Seq a          -- O(n)
fmap   :: (a → Bool) → Seq a → Seq b          -- O(n)

index  :: Seq a → Int → a                     -- O(log(min(i, n − i)))
splitAt :: Seq a → Int → (Seq a, Seq a)       -- O(log(min(i, n − i)))
```

Implementation of sequences

Sequences are implemented as a special form of trees called finger trees:

```
newtype Seq a = Seq (FingerTree a)
data FingerTree a =
      Empty
    | Single a
    | Deep {-# UNPACK #-} !Int
             !(Digit a) (FingerTree (Node a)) !(Digit a)
data Node a = Node2 {-# UNPACK #-} !Int a a
            | Node3 {-# UNPACK #-} !Int a a a
data Digit a = One a | Two a a | Three a a a | Four a a a a
```

Implementation of sequences

Sequences are implemented as a special form of trees called finger trees:

```
newtype Seq a = Seq (FingerTree a)
data FingerTree a =
      Empty
    | Single a
    | Deep {-# UNPACK #-} !Int
           !(Digit a) (FingerTree (Node a)) !(Digit a)
data Node a = Node2 {-# UNPACK #-} !Int a a
            | Node3 {-# UNPACK #-} !Int a a a
data Digit a = One a | Two a a | Three a a a | Four a a a a
```

Stores the size directly like `Map`.

Implementation of sequences

Sequences are implemented as a special form of trees called finger trees:

```
newtype Seq a = Seq (FingerTree a)
data FingerTree a =
     Empty
   | Single a
   | Deep {-# UNPACK #-} !Int
          !(Digit a) (FingerTree (Node a)) !(Digit a)
data Node a = Node2 {-# UNPACK #-} !Int a a
            | Node3 {-# UNPACK #-} !Int a a a
data Digit a = One a | Two a a | Three a a a | Four a a a a
```

Calls itself recursively, but at a different type!

Implementation of sequences

Sequences are implemented as a special form of trees called finger trees:

```
newtype Seq a = Seq (FingerTree a)
data FingerTree a =
      Empty
    | Single a
    | Deep {-# UNPACK #-} !Int
           !(Digit a) (FingerTree (Node a)) !(Digit a)
data Node a = Node2 {-# UNPACK #-} !Int a a
            | Node3 {-# UNPACK #-} !Int a a a
data Digit a = One a | Two a a | Three a a a | Four a a a a
```

These are the first and the last few elements. They're directly accessible.

Implementation of sequences

Sequences are implemented as a special form of trees called finger trees:

```
newtype Seq a = Seq (FingerTree a)
data FingerTree a =
      Empty
    | Single a
    | Deep {-# UNPACK #-} !Int
           !(Digit a) (FingerTree (Node a)) !(Digit a)
data Node a = Node2 {-# UNPACK #-} !Int a a
            | Node3 {-# UNPACK #-} !Int a a a
data Digit a = One a | Two a a | Three a a a | Four a a a a
```

This is an example of a so-called nested datatype.

Arrays

Arrays

Sometimes, we want fast (constant time) access to data and compact storage.

Note that sequences get close (operations at a low logarithmic cost), but when speed and space are really an issue, arrays may be better:

- there are simple arrays provided as part of the `array` package;
- the still relatively recent `vector` package is quickly becoming a new favourite, and that's what we'll discuss here.

Persistent arrays and updates

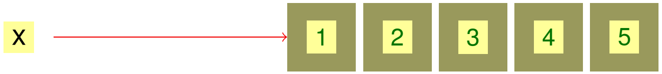
The expression

```
let x = fromList [1, 2, 3, 4, 5] in x // [(2, 13)]
```

evaluates to the same array as `fromList [1, 2, 13, 4, 5]`.

Question: How expensive is the update operation?

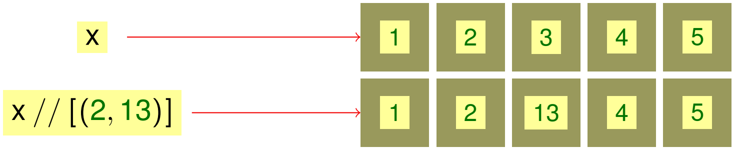
Array updates



$x \mathbin{//} [(2, 13)]$

- 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).

Array updates



- 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 persistent arrays

Be careful when using them:

- stay away if you require a large number of small incremental updates – finite maps or sequences 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.

Vector interface

This is an excerpt of Data.Vector:

```
data Vector a   -- abstract

empty    :: Vector a                           -- O(1)
generate :: Int → (Int → a) → Vector a         -- O(n)
fromList :: [a] → Vector a                     -- O(n)

(++)     :: Vector a → Vector a → Vector a     -- O(m + n)

(!)      :: Vector a → Int → a                 -- O(1)
(!?)     :: Vector a → Int → Maybe a           -- O(1)
slice    :: Int → Int → Vector a → Vector a    -- O(1)

(//)     :: Vector a → [(Int, a)] → Vector a   -- O(m + n)

map      :: (a → b) → Vector a → Vector b      -- O(n)
filter   :: (a → Bool) → Vector a → Vector a   -- O(n)
foldr    :: (a → b → b) → b → Vector a → b     -- O(n)
```

Note the efficient slicing.

Implementation of vectors

- The implementation of the vector library falls back on primitive built-in arrays implemented directly in GHC.
- It additionally stores a lower and upper bound. These are used to implement the slicing operations.

Unboxed types, unboxed vectors

The internals of basic types

```
⟩ :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
```

The internals of basic types

```
⟩ :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
```

Aha, so GHC thinks `Int` is yet another datatype?

- The `GHC.Types` and `GHC.Prim` are just module names.
- So there's one constructor, called `I#`.
- And one argument, of type `Int#`.

The internals of basic types

```
⟩ :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
```

Aha, so GHC thinks `Int` is yet another datatype?

- The `GHC.Types` and `GHC.Prim` are just module names.
- So there's one constructor, called `I#`.
- And one argument, of type `Int#`.

What is an `Int#`?

The internals of basic types (contd.)

To get names like Int# even through the parser, we have to enable the `MagicHash` language extension . . .

```
⟩ :i GHC.Prim.Int#
data GHC.Prim.Int#    -- Defined in ‘ GHC.Prim ’
```

So this one seems to be really primitive.

Boxed vs. unboxed types

The type Int# is the type of unboxed integers:

- unboxed integers are essentially machine integers,
- their memory representation is just bits encoding an integer.

Boxed vs. unboxed types

The type Int# is the type of unboxed integers:

- unboxed integers are essentially machine integers,
- their memory representation is just bits encoding an integer.

An Int is a boxed integer:

- it wraps the unboxed integer in an additional pointer,
- thereby introducing an indirection.

Boxed vs. unboxed types (contd.)

Pro unboxed:

- no indirection,
- faster,
- less space.

Boxed vs. unboxed types (contd.)

Pro unboxed:

- no indirection,
- faster,
- less space.

Pro boxed:

- only boxed types admit laziness,
- only boxed types admit polymorphism.

Boxing makes all types look alike, making it compatible with suspended computations and polymorphism.

Operations on unboxed types

Everything is monomorphic:

```
3#     :: Int#
3##    :: Word#
3.0#   :: Float#
3.0##  :: Double#
'c'#   :: Char#

(+#)       :: Int#    → Int#    → Int#
plusWord#  :: Word#   → Word#   → Word#
plusFloat# :: Float#  → Float#  → Float#
(+##)      :: Double# → Double# → Double#
```

The kind of unboxed types

GHC uses Haskell's kind system to distinguish boxed from unboxed types:

```
Int  :: *
[]   :: * → *
Int# :: #
```

Kinds are the "types" of types:

- where types classify expressions, kinds classify types;
- only well-kinded types are admitted by Haskell;
- the kind and type layers are quite similar, but (further language extensions ignored) there are only `*` and `#` as base kinds, and there is no polymorphism on the kind level.

Kind errors

All these expressions produce kind errors:

```
let x = undefined :: []
3# +# 2
id 3#
[3#]
```

Type variables scope only over kind `*`. The kind system thus prevents polymorphic use of unboxed types.

Advice on unboxed types

You very rarely have to use unboxed types directly:

- GHC's optimizer is quite good at removing some unnecessary boxing and unboxing;
- there are libraries that offer good abstractions of internally unboxed values;
- one can instruct GHC to specifically unbox certain values via UNPACK pragmas.

Unboxed vectors

The vector package provides unboxed vectors next to the regular, boxed ones:

- provided by the `Data.Vector.Unboxed` module;
- more compact and more local storage;
- restricted w.r.t. the element type.

The Unbox class

General user-defined datatypes cannot be unboxed, so somewhat necessarily unboxed vectors are only available for a limited class of element types:

```
class Unbox a where
  ...
```

The Unbox class

General user-defined datatypes cannot be unboxed, so somewhat necessarily unboxed vectors are only available for a limited class of element types:

```
class Unbox a where
  ...
```

```
instance Unbox Int
instance Unbox Float
instance Unbox Double
instance Unbox Char
instance Unbox Bool
instance (Unbox a, Unbox b) ⇒ Unbox (a, b)
```

Interface of unboxed vectors

The module Data.Vector.Unsafe is similar to Data.Vector:

```
data Vector a   -- abstract

empty    :: Unbox a ⇒ Vector a
generate :: Unbox a ⇒ Int → (Int → a) → Vector a
fromList :: Unbox a ⇒ [a] → Vector a

(++) :: Unbox a ⇒ Vector a → Vector a → Vector a

(!)  :: Unbox a ⇒ Vector a → Int → a
(!?) :: Unbox a ⇒ Vector a → Int → Maybe a
slice :: Unbox a ⇒ Int → Int → Vector a → Vector a

(//) :: Unbox a ⇒ Vector a → [(Int, a)] → Vector a

map :: (Unbox a, Unbox b) ⇒ (a → b) → Vector a → Vector b
filter :: Unbox a ⇒ (a → Bool) → Vector a → Vector a
foldr :: Unbox a ⇒ (a → b → b) → b → Vector a → b
```

Complexity as before.

Implementation of unboxed vectors

- While the internals of unboxed vectors are partially built into GHC as well, the outer interface and the Unbox class are actually implemented as a library.
- The library selects an appropriate implementation automatically depending on the type of array element, by means of a datatype family. More on (data)type families tomorrow.

Mutable vectors

Ephemeral data structures in Haskell

Some (surprisingly few, but some) algorithms can be implemented more efficiently in the presence of destructive updates:

- Haskell has mutable data structures as well as immutable ones;
- most operations on mutable data structures have `IO` type.

Ephemeral data structures in Haskell

Some (surprisingly few, but some) algorithms can be implemented more efficiently in the presence of destructive updates:

- Haskell has mutable data structures as well as immutable ones;
- most operations on mutable data structures have `IO` type.

Both `Data.Vector.Mutable` and `Data.Vector.Unboxed.Mutable` export a datatype

```
data IOVector (a :: *)    -- abstract
```

of mutable vectors.

Mutable vector interface

```
new       :: Int → IO (IOVector a)
replicate :: Int → a → IO (IOVector a)
read  ::  IOVector a → Int → IO a
write ::  IOVector a → Int → a → IO ()
...
```

Strings

Haskell strings

By default, Haskell strings are lists of characters:

```
type String = [Char]
```

Haskell strings

By default, Haskell strings are lists of characters:

```
type String = [Char]
```

This definition is quite convenient for implementing basic text processing functions, as one can reuse the rich libraries for lists, but the list representation is quite inefficient for dealing with large amounts of text.

Question

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

ByteString and Text

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 also deals with encoding issues.
- Both export an interface quite similar to that of lists or vectors with many of the familiar functions.
- There are specific lazy variants of Text and ByteString that are suitable if huge strings are to be processed as streams that should not be held in memory completely.

More data structures 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 – in particular array updates – 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.