Data Structures

Haskell Performance

Andres Löh

14–15 May 2018 —

Kinds

What is `Maybe` ?

The type signature of undefined is

```
undefined :: a
```

indicating that `undefined` can have any type.

What is `Maybe` ?

The type signature of undefined is

```
undefined :: a
```

indicating that `undefined` can have any type.

Yet writing

```
undefined :: Maybe
```

yields an error. Why?

What is `Maybe` ?

The type signature of undefined is

```
undefined :: a
```

indicating that `undefined` can have any type.

Yet writing

```
undefined :: Maybe
```

yields an error. Why?

Because `Maybe` always expects a type parameter?

An interesting datatype

```haskell
data WrappedInt f = Wrap (f Int)

example1 :: WrappedInt Maybe
example1 = Wrap (Just 3)

example2 :: WrappedInt []
example2 = Wrap [1, 2, 3]

example3 :: WrappedInt IO
example3 = Wrap readLn
```

Here, `Maybe` can occur without a type parameter.

An interesting datatype

```haskell
data WrappedInt f = Wrap (f Int)

example1 :: WrappedInt Maybe
example1 = Wrap (Just 3)

example2 :: WrappedInt []
example2 = Wrap [1, 2, 3]

example3 :: WrappedInt IO
example3 = Wrap readLn
```

Here, `Maybe` can occur without a type parameter.

What happens if we type the following:

```haskell
Wrap (Just 3) :: WrappedInt (Maybe Int)
```

A kind error

```
GHCi> Wrap (Just 3) :: WrappedInt (Maybe Int)
```

Expecting one fewer argument to ‘`Maybe Int`’

Expected kind ‘`* -> *`’, but ‘`Maybe Int`’ has kind `*`

Kinds are the types of types

Types classify Haskell expressions (and, in a way, also patterns and declarations). Only well-typed expressions are admissible.

Kinds are the types of types

Types classify Haskell expressions (and, in a way, also patterns and declarations). Only well-typed expressions are admissible.

Kinds classify Haskell types. Only well-kinded types are admissible.

The kind `*`

The most important kind in Haskell is called `*` :

- nearly all expressions in Haskell have types that have kind `*` ;
- in particular, if you define an unparameterized datatype using `data` , it has kind `*` ;
- for now, think of kind `*` as the kind of potentially inhabited types, or as the kind of “fully applied” types.

Examples of types of kind `*`

```
Int
Char
Bool
```

Examples of types of kind `*`

```
Int
Char
Bool
```

```
Maybe Int
Int -> Int
[[[Char]]]
(Bool, Char -> [Ordering -> IO ()])
WrappedInt Maybe
a -> b -> a
```

Function kinds

You can see `Maybe` as a function on the type level: it expects an argument which is a type, and returns a new type.

Function kinds

You can see `Maybe` as a function on the type level: it expects an argument which is a type, and returns a new type.

But can we apply anything to `Maybe` ? What about the following "type"?

```
Maybe IO
```

We actually want that the argument to `Maybe` is a potentially inhabited type, i.e., a type of kind `*` . We then obtain a new type of kind `*` .

Function kinds

You can see `Maybe` as a function on the type level: it expects an argument which is a type, and returns a new type.

But can we apply anything to `Maybe` ? What about the following "type"?

```
Maybe IO
```

We actually want that the argument to `Maybe` is a potentially inhabited type, i.e., a type of kind `*` . We then obtain a new type of kind `*` .

This motivates:

```
Maybe :: * -> *
```

Kinds in GHCi

- Haskell can infer kinds, just as it can infer types.
- And you can ask GHCi for the inferred kind of a type.
- To obtain the inferred kind for a type term, type `:k` or `:kind` followed by the term at the GHCi prompt.

Kinds of parameterized types

```
Maybe :: * -> *
IO    :: * -> *
[]    :: * -> *
(,)   :: * -> * -> *
(,,)  :: * -> * -> * -> *
(->)  :: * -> * -> *
State :: * -> * -> *
```

Kinds of parameterized types (contd.)

Note that lists, tuples and functions despite their built-in syntax actually all support a prefix notation on the type level. Writing

```
(->) Int ([] Bool)
```

is equivalent to

```
Int -> [Bool]
```

Kind errors

We can now determine why

```
undefined :: Maybe
undefined :: Maybe IO
```

are wrong.

Kind errors

We can now determine why

```
undefined :: Maybe
undefined :: Maybe IO
```

are wrong.

A type signature attached to a term is expected to be of kind `*` , but `Maybe` without an argument is of kind `* -> *` .

Kind errors

We can now determine why

```
undefined :: Maybe
undefined :: Maybe IO
```

are wrong.

A type signature attached to a term is expected to be of kind `*` , but `Maybe` without an argument is of kind `* -> *` .

A `Maybe` expects an argument of kind `*` , but `IO` is of kind `* -> *` .

Back to `WrappedInt`

Question: What is the kind of `WrappedInt` ?

```
data WrappedInt f = Wrap (f Int)
```

Back to `WrappedInt`

Question: What is the kind of `WrappedInt` ?

```
data WrappedInt f = Wrap (f Int)
```

```
WrappedInt :: (* -> *) -> *
```

Back to `WrappedInt`

Question: What is the kind of `WrappedInt` ?

```
data WrappedInt f = Wrap (f Int)
```

```
WrappedInt :: (* -> *) -> *
```

Thus:

```
WrappedInt Maybe          -- kind correct
WrappedInt (Maybe Int)    -- kind error
```

Kind signatures

- The kind system is a part of the Haskell Standard, but as defined, kinds are completely hidden from the surface and do not occur explicitly in the language.
- However, you can enable explicit kind signatures via the `KindSignatures` language extension.

Kind signatures

- The kind system is a part of the Haskell Standard, but as defined, kinds are completely hidden from the surface and do not occur explicitly in the language.
- However, you can enable explicit kind signatures via the `KindSignatures` language extension.

Example:

```
data WrappedInt (f :: * -> *) = Wrap (f Int :: *)
```

Kind annotations are possible for arguments and type terms, but there's no separate "kind signature" declaration as there is for functions and constants.

Kinds and classes

Type classes are parameterized by types of a particular kind:

```
class Eq a where
  (==) :: a -> a -> Bool  -- a of kind *
  ...
class Functor f where
  fmap :: (a -> b) -> f a -> f b  -- f of kind * -> *
```

Kinds and classes

Type classes are parameterized by types of a particular kind:

```
class Eq a where
  (==) :: a -> a -> Bool  -- a of kind *
  ...
class Functor f where
  fmap :: (a -> b) -> f a -> f b  -- f of kind * -> *
```

Again, with `KindSignatures`, you could write more explicitly:

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

The kind `Constraint`

With the `ConstraintKinds` language extension, we are allowed to talk about the kind `Constraint` of class constraints explicitly:

```
Eq          :: * -> Constraint
Show        :: * -> Constraint

Functor     :: (* -> *) -> Constraint
Applicative :: (* -> *) -> Constraint
Monad       :: (* -> *) -> Constraint
```

Partial parameterization of types

Just like functions are typically curried in Haskell, types are too.

Types can be – and often are – partially parameterized:

```
instance Monad (Either e) where
  ...
```

Partial parameterization of types

Just like functions are typically curried in Haskell, types are too.

Types can be – and often are – partially parameterized:

```
instance Monad (Either e) where
  ...
```

```
Either   :: * -> * -> *
Either e :: * -> *  -- correct kind for Monad class
```

More examples

```
instance Functor ((,) t) where

```

More examples

```
instance Functor ((,) t) where
```

The specialized type of `fmap` must be:

```
fmap :: (a -> b) -> ((,) t) a -> ((,) t) b
```

or syntactically simplified

```
fmap :: (a -> b) -> (t, a) -> (t, b)
```

More examples

```haskell
instance Functor ((,) t) where
  fmap f (x, y) = (x, f y)
```

More examples

```
instance Functor ((,) t) where
  fmap f (x, y) = (x, f y)
```

And what about this one?

```
fmap' :: (a -> b) -> (a, t) -> (b, t)
fmap' f (x, y) = (f x, y)
```

No “type-level lambda”

While we can define

```
fmap' :: (a -> b) -> (a, t) -> (b, t)
fmap' f (x, y) = (f x, y)
```

we cannot make this function be a normal `fmap` on pairs.

No “type-level lambda”

While we can define

```
fmap' :: (a -> b) -> (a, t) -> (b, t)
fmap' f (x, y) = (f x, y)
```

we cannot make this function be a normal `fmap` on pairs.

- In general, there is no easy way to partially apply types to anything but the initial argument(s).
- Unlike for functions, there is no type-level `flip` function.
- The order of arguments of multi-argument datatypes is sometimes carefully chosen in order to admit certain class instances.

Type synonyms do not help

Question: Why does

```
type Flip f a b = f b a
```

```
class Functor (Flip (,) t) where
  fmap = fmap'
```

not work?

Type synonyms do not help

Question: Why does

```
type Flip f a b = f b a
```

```
class Functor (Flip (,) t) where
  fmap = fmap'
```

not work?

Because type synonyms have to be fully applied.

Type synonyms do not help

Question: Why does

```
type Flip f a b = f b a
```

```
class Functor (Flip (,) t) where
  fmap = fmap'
```

not work?

Because type synonyms have to be fully applied.

Note

- how allowing this would make the job of resolving class constraints much harder than it already is;
- that `Flip` itself is a legal type synonym. What is its kind?

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]
```

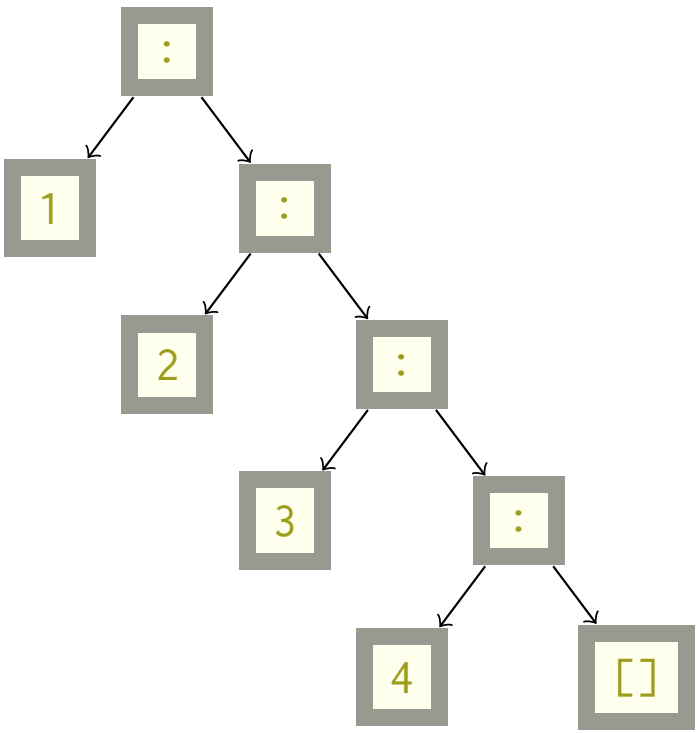
Example: Haskell lists

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

Example: Haskell lists

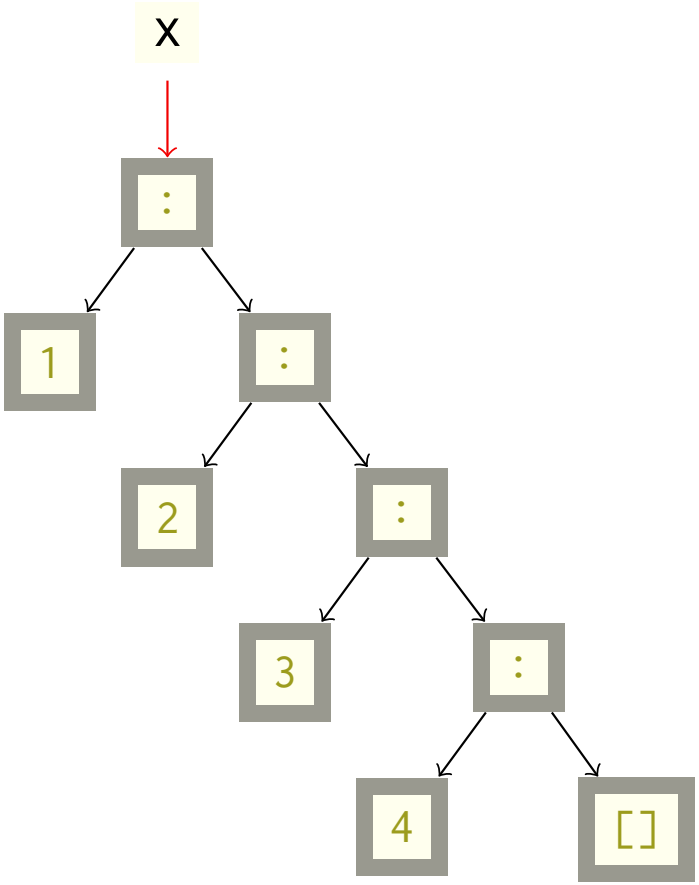
`[1, 2, 3, 4]` is syntactic sugar for `1 : (2 : (3 : (4 : [])))`

Representation in memory:



Lists are persistent

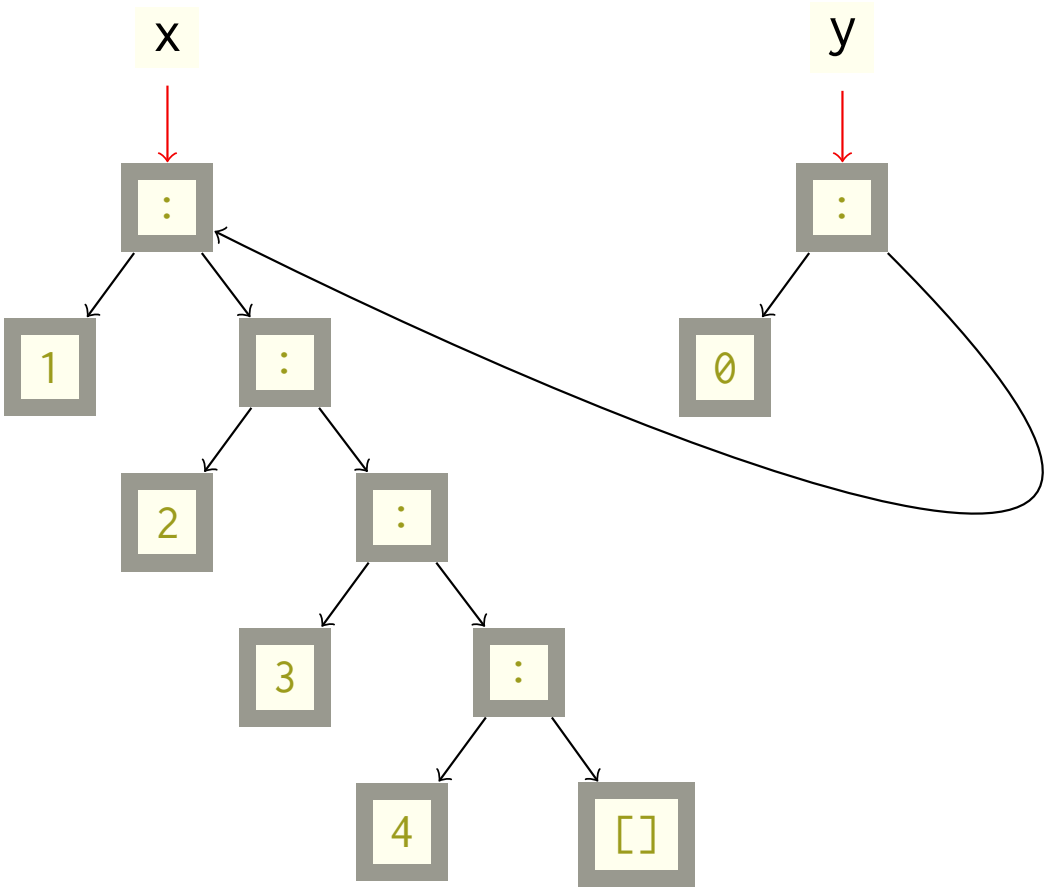
```
let x = [1, 2, 3, 4]; y = 0 : x; z = drop 3 y in...
```



Well-Typed

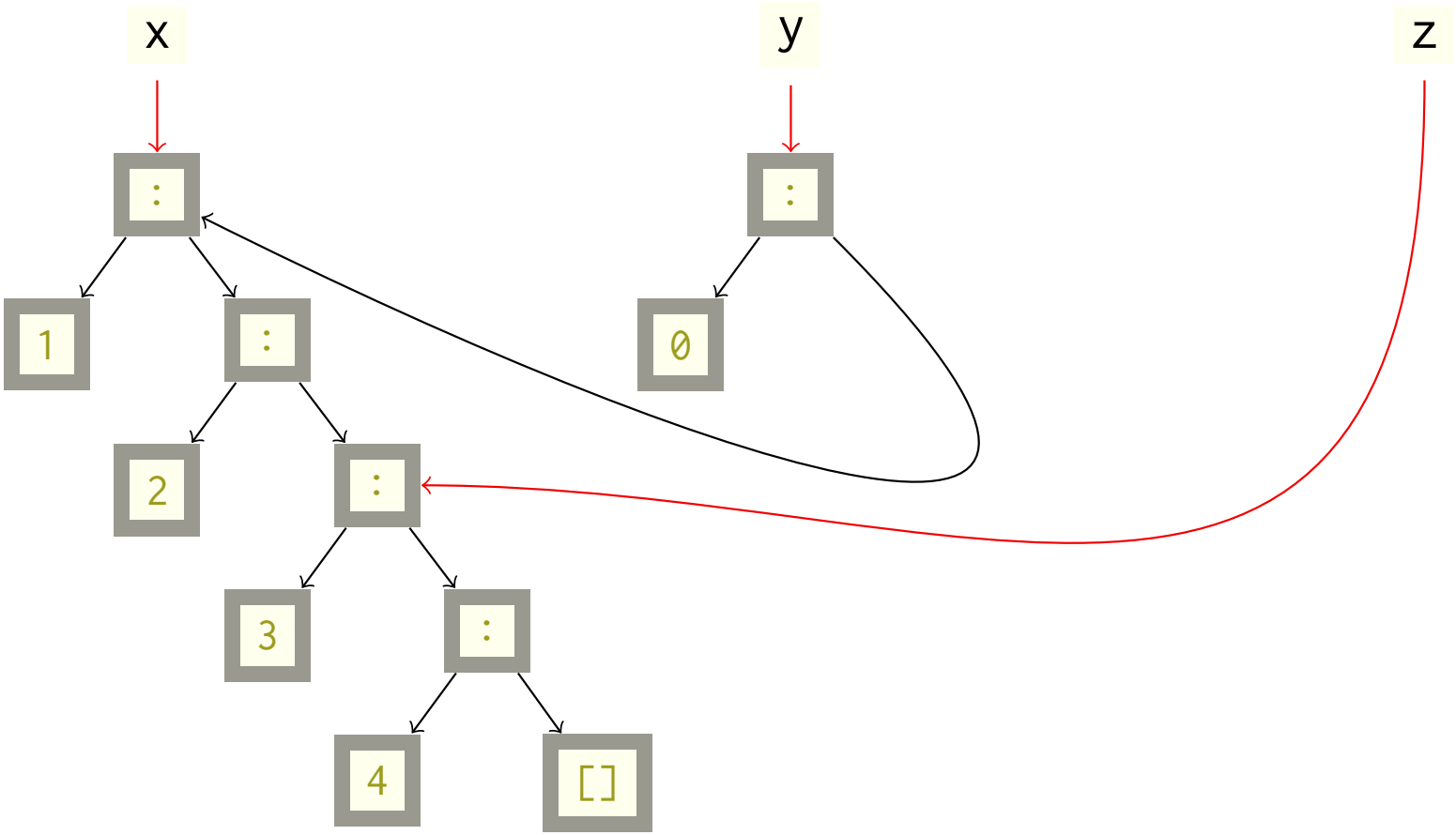
Lists are persistent

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



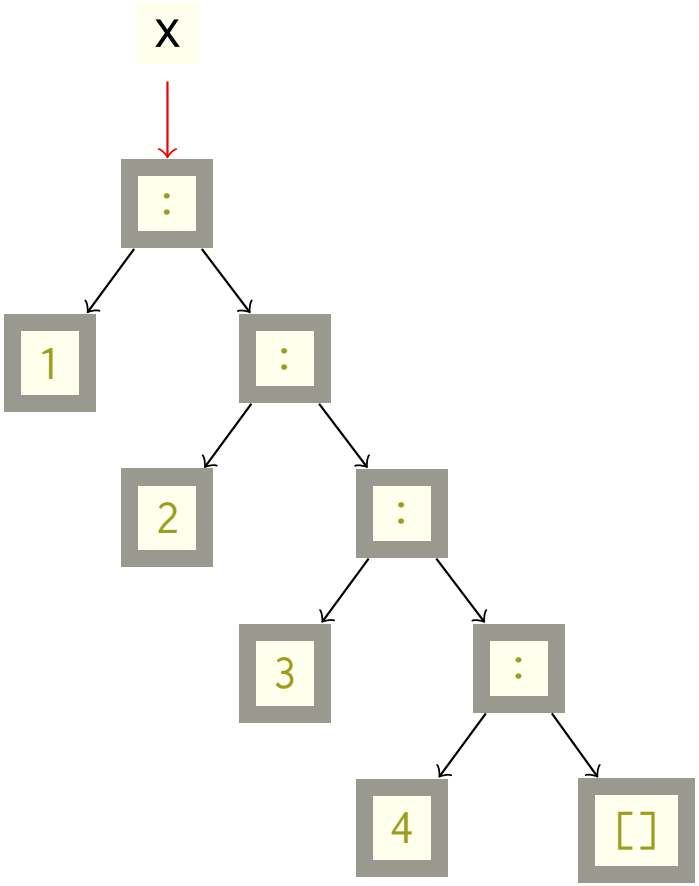
Lists are persistent

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



Lists are persistent (contd.)

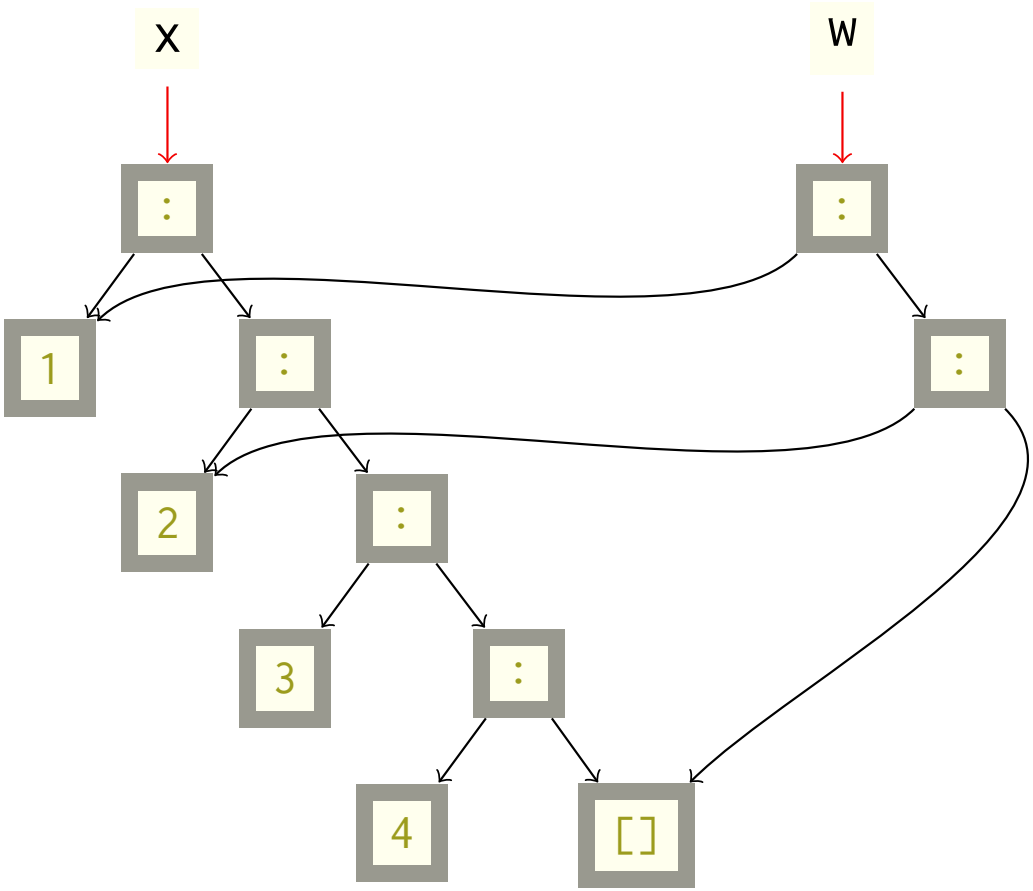
```
let x = [1, 2, 3, 4]; w = take 2 x in...
```



Well-Typed

Lists are persistent (contd.)

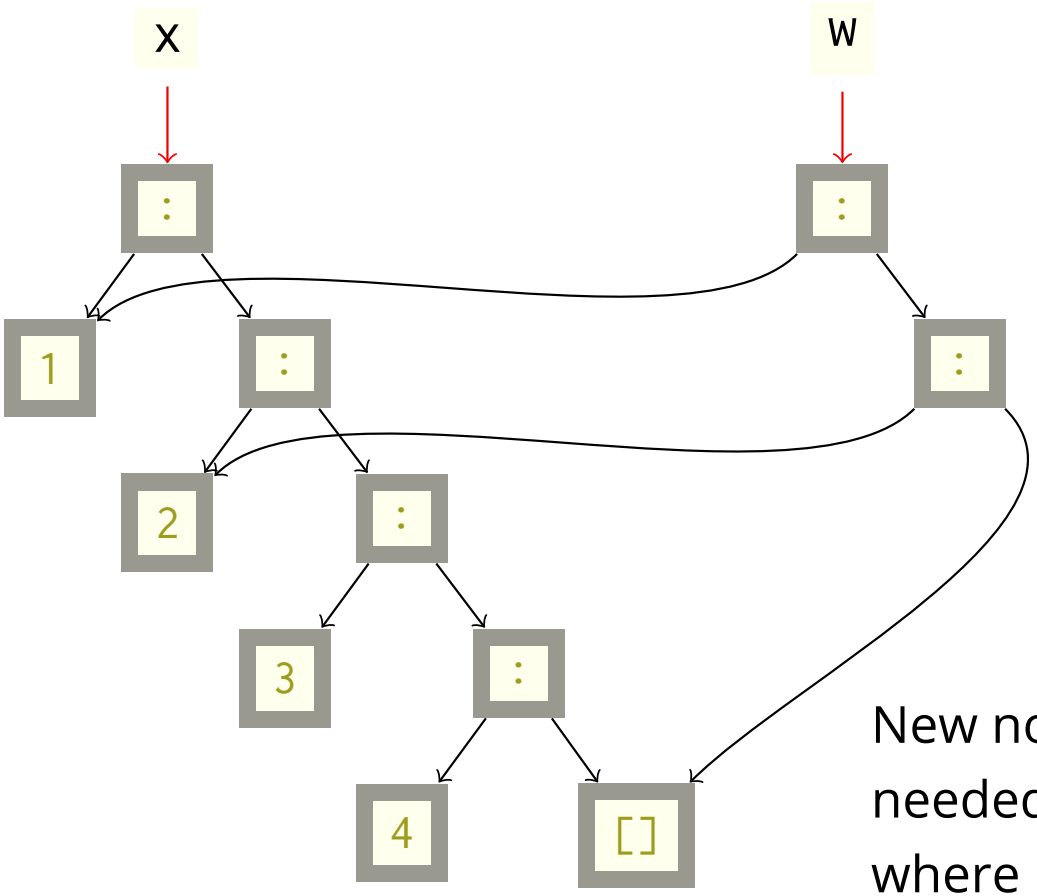
```
let x = [1, 2, 3, 4]; w = take 2 x in...
```



Well-Typed

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

```
:vis
```

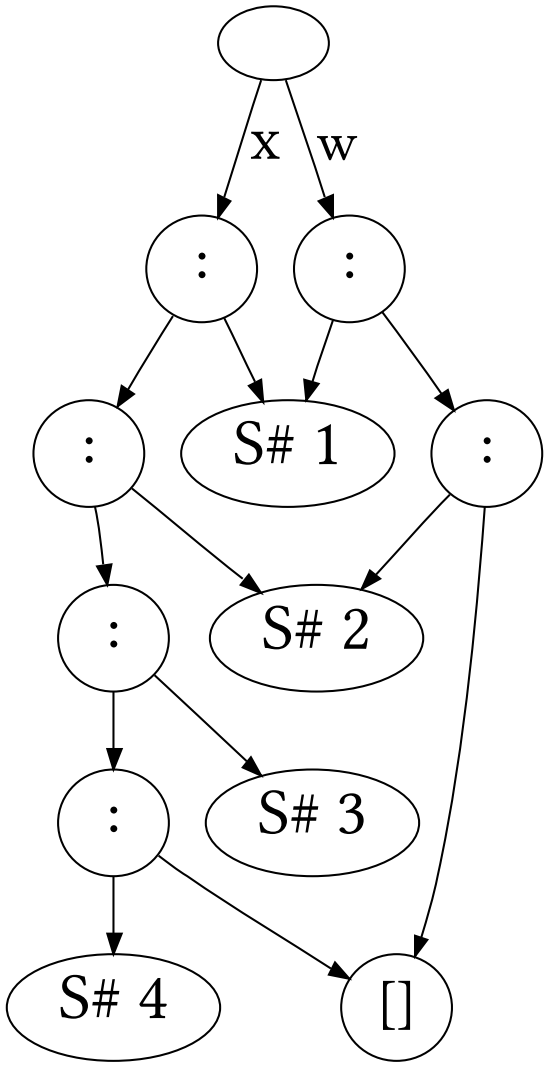
Add terms to view; switch to graph view:

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

Evaluate `y` and update:

```
GHCi> y
[1, 2]
GHCi> :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]                          --
(:)     :: a -> [a] -> [a]              --
head    :: [a] -> a                     --
tail    :: [a] -> [a]                   --
snoc    :: [a] -> a -> [a]              --
snoc     = \ xs x -> xs ++ [x]
(!!)    :: [a] -> Int -> a              --
(++)    :: [a] -> [a] -> [a]            --
reverse :: [a] -> [a]                   --
splitAt :: Int -> [a] -> ([a], [a])     --
union   :: Eq a => [a] -> [a] -> [a]    --
elem    :: Eq a => a -> [a] -> Bool     --
```

Operations on lists

```
[]      :: [a]                          -- O(1)
(:)     :: a -> [a] -> [a]              --
head    :: [a] -> a                     --
tail    :: [a] -> [a]                   --
snoc    :: [a] -> a -> [a]              --
snoc     = \ xs x -> xs ++ [x]
(!!)    :: [a] -> Int -> a              --
(++)    :: [a] -> [a] -> [a]            --
reverse :: [a] -> [a]                   --
splitAt :: Int -> [a] -> ([a], [a])     --
union   :: Eq a => [a] -> [a] -> [a]    --
elem    :: Eq a => a -> [a] -> Bool     --
```

Operations on lists

```
[]      :: [a]                          -- O(1)
(:)     :: a -> [a] -> [a]              -- O(1)
head    :: [a] -> a                     --
tail    :: [a] -> [a]                   --
snoc    :: [a] -> a -> [a]              --
snoc     = \ xs x -> xs ++ [x]
(!!)    :: [a] -> Int -> a              --
(++)    :: [a] -> [a] -> [a]            --
reverse :: [a] -> [a]                   --
splitAt :: Int -> [a] -> ([a], [a])     --
union   :: Eq a => [a] -> [a] -> [a]    --
elem    :: Eq a => a -> [a] -> Bool     --
```

Operations on lists

```
[]      :: [a]                          -- O(1)
(:)     :: a -> [a] -> [a]              -- O(1)
head    :: [a] -> a                     -- O(1)
tail    :: [a] -> [a]                   --
snoc    :: [a] -> a -> [a]              --
snoc     = \ xs x -> xs ++ [x]
(!!)    :: [a] -> Int -> a              --
(++)    :: [a] -> [a] -> [a]            --
reverse :: [a] -> [a]                   --
splitAt :: Int -> [a] -> ([a], [a])     --
union   :: Eq a => [a] -> [a] -> [a]    --
elem    :: Eq a => a -> [a] -> Bool     --
```

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]              --
snoc     = \ xs x -> xs ++ [x]
(!!)    :: [a] -> Int -> a              --
(++)    :: [a] -> [a] -> [a]            --
reverse :: [a] -> [a]                   --
splitAt :: Int -> [a] -> ([a], [a])     --
union   :: Eq a => [a] -> [a] -> [a]    --
elem    :: Eq a => a -> [a] -> Bool     --
```

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    = \ xs x -> xs ++ [x]
(!!)    :: [a] -> Int -> a              --
(++)    :: [a] -> [a] -> [a]            --
reverse :: [a] -> [a]                   --
splitAt :: Int -> [a] -> ([a], [a])     --
union   :: Eq a => [a] -> [a] -> [a]    --
elem    :: Eq a => a -> [a] -> Bool     --
```

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    = \ xs x -> xs ++ [x]
(!!)    :: [a] -> Int -> a              -- O(n)
(++)    :: [a] -> [a] -> [a]            --
reverse :: [a] -> [a]                   --
splitAt :: Int -> [a] -> ([a], [a])     --
union   :: Eq a => [a] -> [a] -> [a]    --
elem    :: Eq a => a -> [a] -> Bool     --
```

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    = \ xs x -> xs ++ [x]
(!!)    :: [a] -> Int -> a              -- O(n)
(++)    :: [a] -> [a] -> [a]            -- O(m) (first list)
reverse :: [a] -> [a]                   --
splitAt :: Int -> [a] -> ([a], [a])     --
union   :: Eq a => [a] -> [a] -> [a]    --
elem    :: Eq a => a -> [a] -> Bool     --
```

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    = \ xs 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])     --
union   :: Eq a => [a] -> [a] -> [a]    --
elem    :: Eq a => a -> [a] -> Bool     --
```

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    = \ xs 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]    --
elem    :: Eq a => a -> [a] -> Bool     --
```

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    = \ xs 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     --
```

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    = \ xs 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 in the tree are ordered, so that efficient lookup is possible.
- Requires the keys to be in `Ord` .
- 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
```

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.

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.

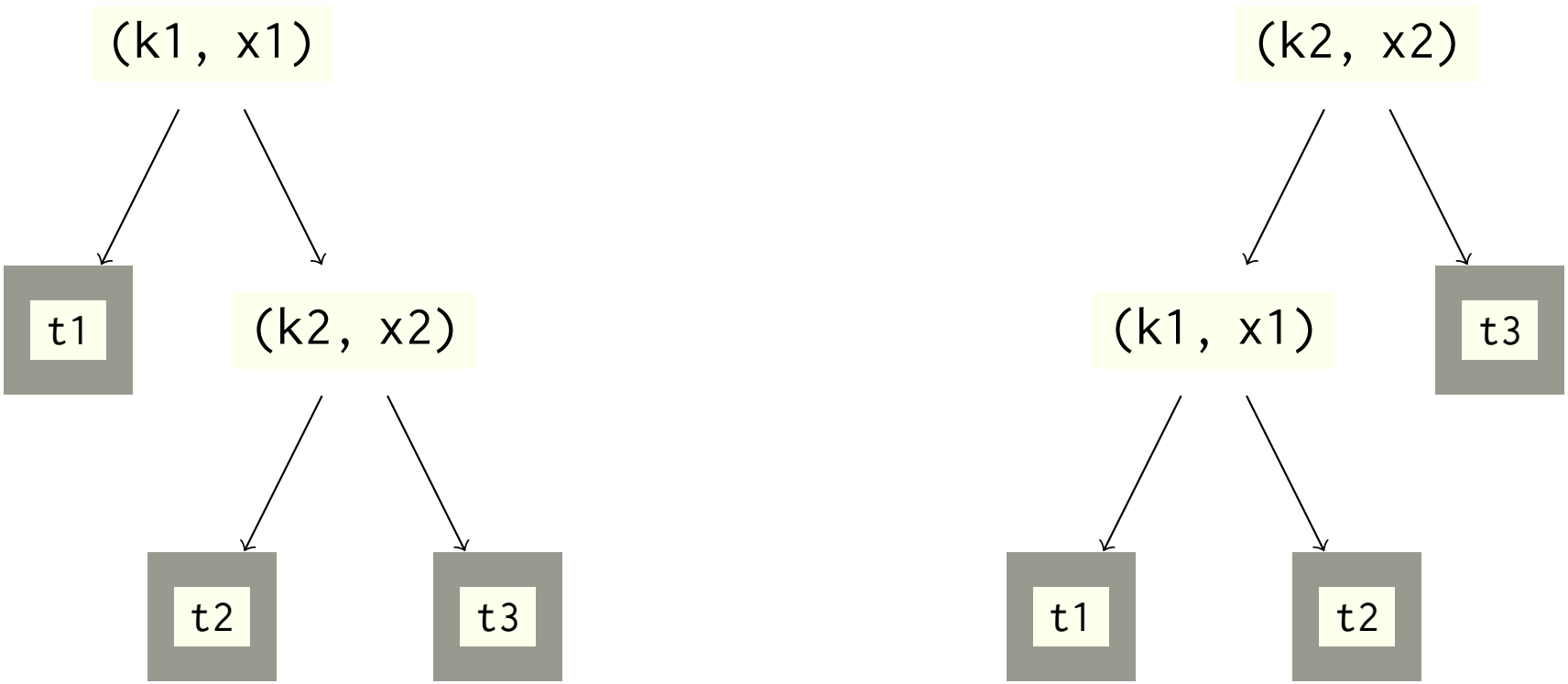
Rotation

```
rotateL :: Map k a -> k -> a -> Map k a
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.

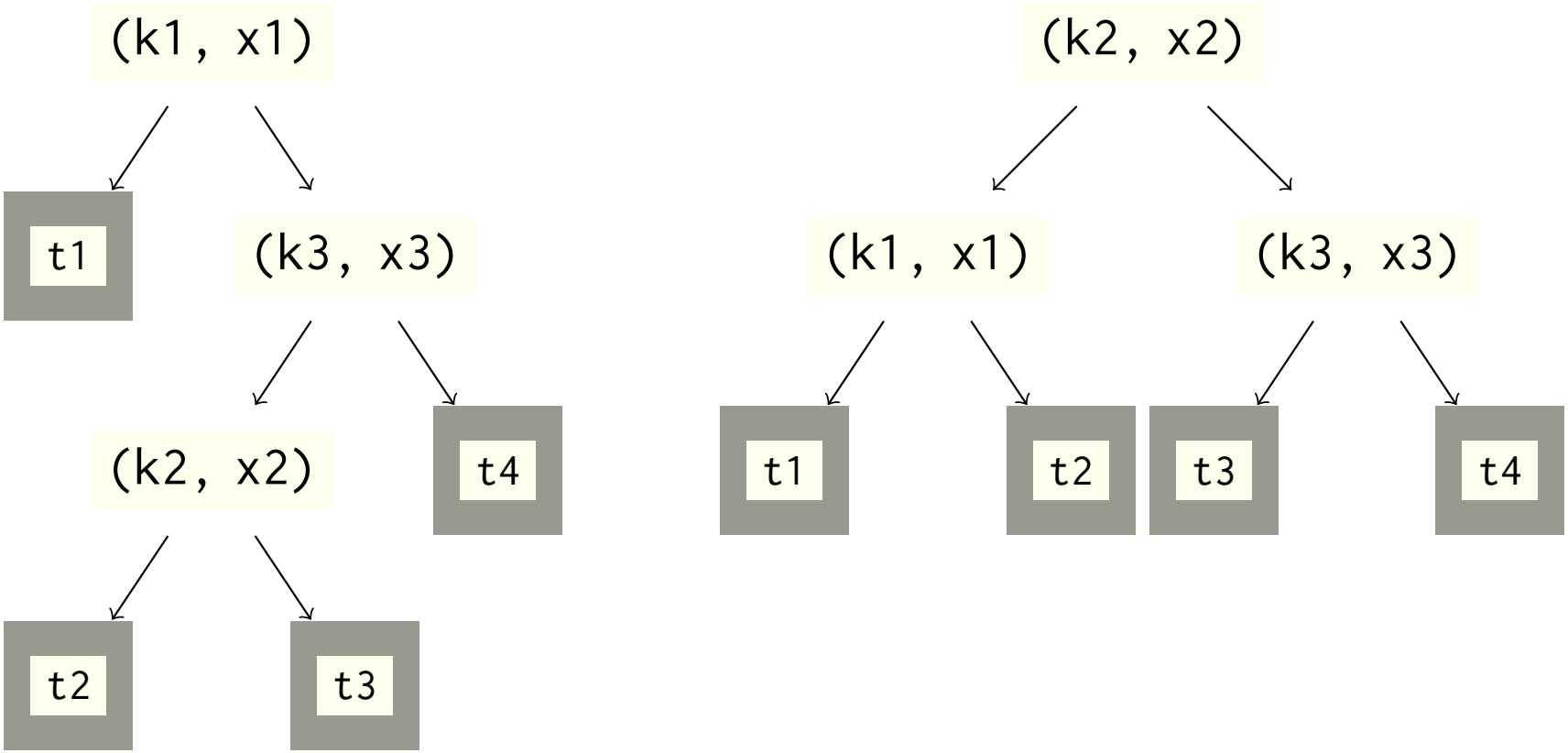
singleL

```
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

```
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)
```



Well-Typed

Rotation (contd.)

Note

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]` .

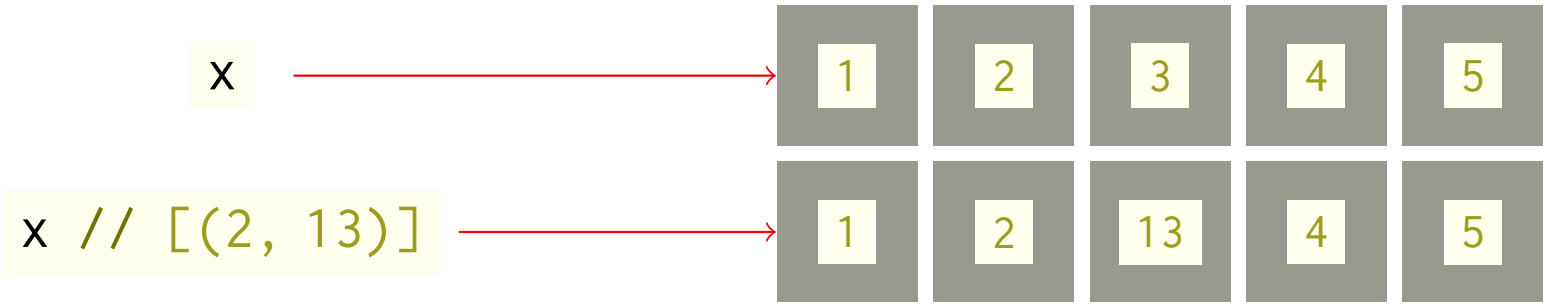
Question: How expensive is the update operation?

Array updates


```
x // [(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

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

The internals of basic types

```
GHCi> :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

```
GHCi> :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 ...

```
GHCi> :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,
- normal polymorphism scopes only over boxed types.

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

Operations on unboxed types

```
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#
```

Type representations

The kind of unboxed types

```
GHCi> :k Int#
Int# :: TYPE 'IntRep
GHCi> :k Word#
Word# :: TYPE 'WordRep
GHCi> :k Double#
Double# :: TYPE 'DoubleRep
```

The kind of unboxed types

```
GHCi> :k Int#
Int# :: TYPE 'IntRep
GHCi> :k Word#
Word# :: TYPE 'WordRep
GHCi> :k Double#
Double# :: TYPE 'DoubleRep
```

Actually, `*` is a synonym for `TYPE 'PtrRepLifted`.

The kind of unboxed types

```
GHCi> :k Int#
Int# :: TYPE 'IntRep
GHCi> :k Word#
Word# :: TYPE 'WordRep
GHCi> :k Double#
Double# :: TYPE 'DoubleRep
```

Actually, `*` is a synonym for `TYPE 'PtrRepLifted`.

You can think of `IntRep`, `WordRep`, and `DoubleRep` and `PtrRepLifted` as just kind parameters.

Polymorphism and kinds

Normal polymorphism restricts the kind to lifted types:

```
id :: a -> a
id x = x
```

is actually an abbreviated form of

```
id :: forall (a :: *) . a -> a
id x = x
```

(which requires `ExplicitForAll` and `KindSignatures`).

Kind errors

All these expressions produce kind errors:

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

The kind system prevents polymorphic use of unboxed types.

GHCi

```
GHCi> id 3#
Couldn't match lifted type with an unlifted type
When matching the kind of 'GHC.Prim.Int#'
```

GHCi

```
GHCi> id 3#
Couldn't match lifted type with an unlifted type
When matching the kind of 'GHC.Prim.Int#'
```

Nearly all polymorphic functions are only applicable to normal, lifted types.

Levity polymorphism

A select few functions have “levity-polymorphic” types:

```
GHCi> :i ($)
($) ::
  forall (r :: RuntimeRep) a (b :: TYPE r) .
  (a -> b) -> a -> b
```

Function application is compatible with unlifted types if the function returns an unlifted type.

Example

```
unpackInt :: Int -> Int#
unpackInt (I# x) = x
```

This works due to levity-polymorphic `($)` :

```
GHCi> I# (unpackInt $ 3)
3
```

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

`Data.Vector.Unboxed` 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 later.

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.

Mutable vectors

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?

Size of a linked list

- Each cons-cell is three words long (info table, two pointers for the payload).
- Each character is boxed, and the box is three words long (info table, two words for the character).
- The empty list is one word (info table, no payload).

Size of a linked list

- Each cons-cell is three words long (info table, two pointers for the payload).
- Each character is boxed, and the box is three words long (info table, two words for the character).
- The empty list is one word (info table, no payload).

So all in all 16 words (or 128 bytes on a 64-bit machine).

Size of a linked list

- Each cons-cell is three words long (info table, two pointers for the payload).
- Each character is boxed, and the box is three words long (info table, two words for the character).
- The empty list is one word (info table, no payload).

So all in all 16 words (or 128 bytes on a 64-bit machine).

In fairness, the empty list is shared, and some frequently used characters may be shared as well, but still …

`Text`

The `text` package offers

```
data Text   -- abstract
```

a **packed** representation of (Unicode) text.

- Much less memory overhead than a `String` .
- Still uses UTF-16 encoding per character, so 2 bytes per character.
- Performance characteristics more like functional arrays.

Converting between `String` and `Text`

From `Data.Text` :

```
pack   :: String -> Text  -- O(n)
unpack :: Text -> String  -- O(n)
```

Some common operations

```
cons    :: Char -> Text -> Text                  -- O(n)
(<>)    :: Text -> Text -> Text                  -- O(n)
length  :: Text -> Int                           -- O(n)
map     :: (Char -> Char) -> Text -> Text        -- O(n)
filter  :: (Char -> Bool) -> Text -> Text        -- O(n)
foldr   :: (Char -> a -> a) -> a -> Text -> a    -- O(n)
toUpper :: Text -> Text                          -- O(n)
strip   :: Text -> Text                          -- O(n)
lines   :: Text -> [Text]                        -- O(n)
head    :: Text -> Char                          -- O(1)
last    :: Text -> Char                          -- O(1)
index   :: Text -> Int -> Char                   -- O(n)
```

Stream fusion

Text makes use of **stream fusion** internally:

- some subsequent traversals, such as multiple `map` s followed by a `fold` , will be fused together, so that only a single traversal is required.

Overloaded string literals

With the `OverloadedStrings` language extension, string literals become overloaded.

Without:

```
GHCi> :t "foo"
"foo" :: [Char]
```

With:

```
GHCi> :t "foo"
"foo" :: IsString t => t
```

Lazy text

The same package also offers a module `Data.Text.Lazy` with another

```
data Text   -- abstract
```

The internal representation is a linked list of **chunks**, which are strict text values.

For streaming purposes, not the entire text has to be present in memory at once.

Conversion from/to lazy text

Assuming `Data.Text.Lazy` is available qualified as `Lazy` :

```
toStrict   :: Lazy.Text -> Text
fromStrict :: Text -> Lazy.Text
```

Both `Text` types are an instance of the `IsString` class.

Building text

- Appending text values is not very efficient.
- However, often the building and inspecting phases are separate.
- Then it's useful to build text using `Builder`, and convert it to text after building is complete.
- Intuitively a builder simply allocates a buffer and fills it with incoming data.

Builder interface

Defined in `Data.Text.Lazy.Builder` :

```
data Builder  -- abstract

toLazyText   :: Builder -> Lazy.Text           -- O(n)
fromText     :: Text -> Builder                -- O(1)
fromLazyText :: Lazy.Text -> Builder           -- O(1)
(<>)         :: Builder -> Builder -> Builder  -- O(1)
```

ByteString

Available in the `bytestring` package.

```
data ByteString  -- abstract
```

A bytestring is a **packed** representation of a sequence of bytes.

- Again, much less memory overhead than a `String` .
- Less overhead even than `Text` .
- No interpretation of the characters (no encoding).
- Only useful for ASCII formats or (better) binary data.

Variants of `ByteString`

- Like `Text`, `ByteString` comes with a strict and a lazy variant.
- Like `Text`, both `ByteString` types are an instance of `IsString`.
- Like `Text`, `ByteString` has a `Builder` type.

Conversion between `Text` and `ByteString`

As `ByteString` consists of pure bytes, but `Text` is interpreted, we need an **encoding** in order to translate between the two.

From `Data.Text.Encoding`:

```
encodeUtf8 :: Text -> ByteString
decodeUtf8 :: ByteString -> Text  -- partial!
```

Decoding can fail

Not all byte sequences are valid UTF-8 encodings.

```
GHCi> decodeUtf8 "x\255"
"*** Exception: Cannot decode byte ...
```

Decoding can fail

Not all byte sequences are valid UTF-8 encodings.

```
GHCi> decodeUtf8 "x\255"
"*** Exception: Cannot decode byte ...
```

If you want to be robust against invalid inputs, you either have to catch exceptions or use `decodeUtf8'` :

```
GHCi> :t decodeUtf8'
decodeUtf8' ::
  ByteString -> Either UnicodeException Text
GHCi> isRight $ decodeUtf8' "x\255"
False
```

Conversion between `String` and `ByteString`

Defined in `Data.Text.Char8` :

```
pack   :: String -> ByteString  -- unsafe
unpack :: ByteString -> String  -- unsafe
```

Note that these will only work correctly on the ASCII subset of characters, so these should be used with extreme care.

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.
- Lists are ok for stack-like use or simple traversals.
- Good general-purpose data structures are sets, finite maps and sequences.
- Arrays – in particular array updates – should be used with care.
- When dealing with textual data, `Text` and `ByteString` can be much more efficient than `String` .