Data structures

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK
Dr. Polina Vinogradova, IOHK

2019-02-08

Goals

- Kinds
- Persistent data structures
- Maps
- Sequences
- Nested datatypes

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 (cntd.)

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

```haskell
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`?

```haskell
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:

```haskell
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

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

```

The specialized type of `fmap` must be:

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

or syntactically simplified

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

More examples

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

More examples

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

And what about this one?

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

No "type-level lambda"

While we can define

```haskell
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

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

```haskell
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

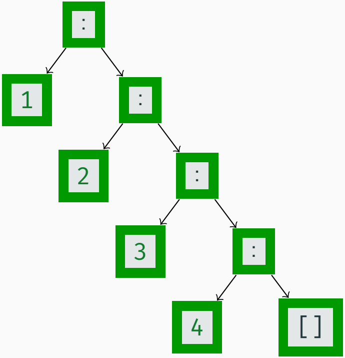
`[1, 2, 3, 4]` is syntactic sugar for

`1 : (2 : (3 : (4 : [])))`

Example: Haskell lists

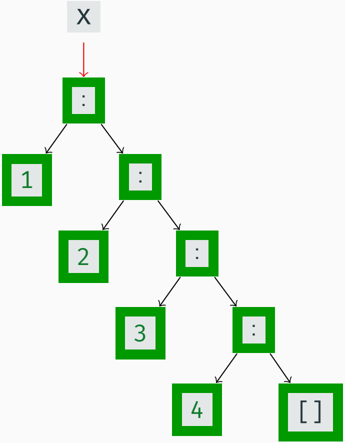
`[1, 2, 3, 4]` is syntactic sugar for
`1 : (2 : (3 : (4 : [])))`

Representation in memory:



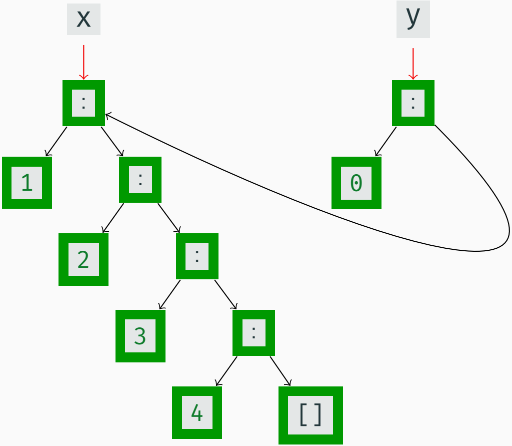
Lists are persistent

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



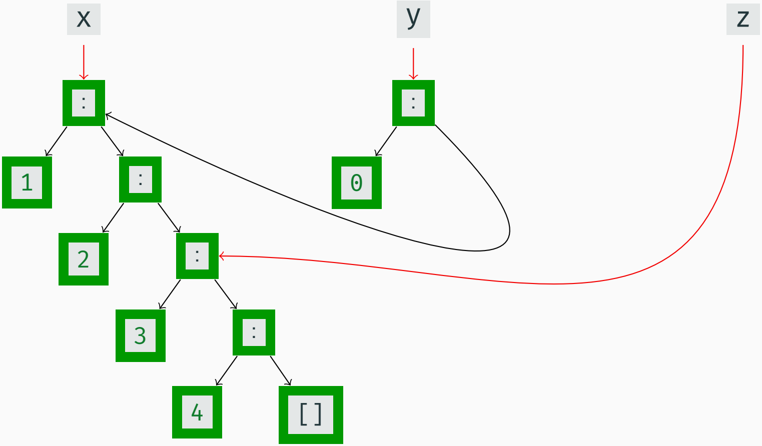
Lists are persistent

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



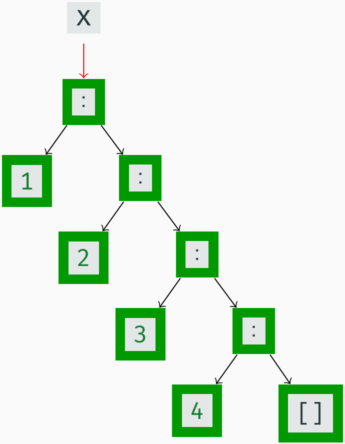
Lists are persistent

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



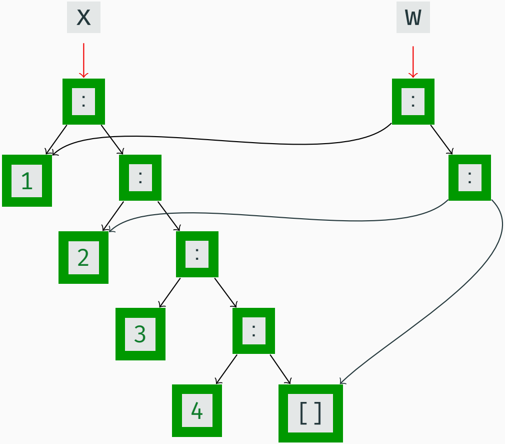
Lists are persistent (contd.)

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



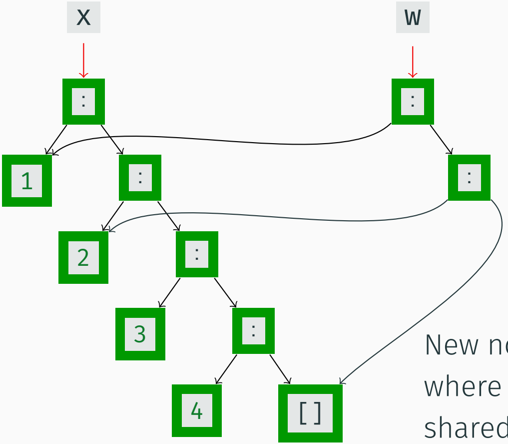
Lists are persistent (contd.)

```
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

```
:vis
```

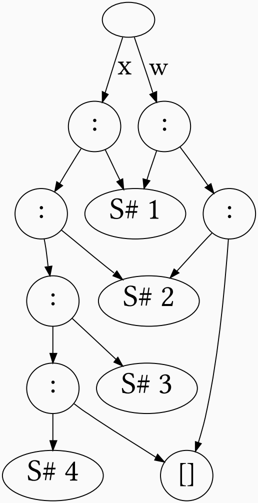
Add terms to view; switch to graph view:

```
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

```haskell
[]      :: [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 are a special case of finite maps: essentially,

```haskell
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`:

```haskell
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

```haskell
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

```haskell
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

```haskell
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

```haskell
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.

Creating finite maps

```haskell
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

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.

Finding an element

```haskell
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

```haskell
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

```haskell
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`:

```haskell
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.

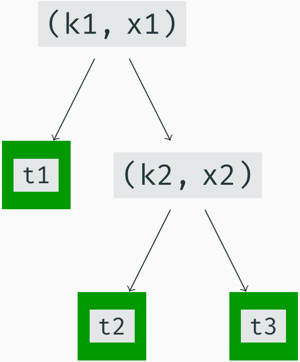
Rotation

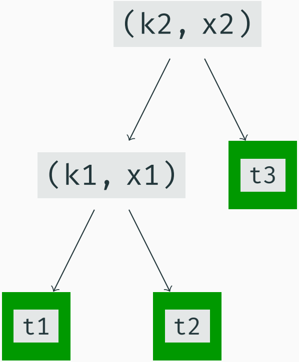
```
rotateL :: Map k a -> k -> a -> Map k a -> Map k a
rotateL l kx x r@(Bin _ ly _ _ ry)
   | size ly < ratio * size ry = singleL l kx x r
   | otherwise                 = doubleL l kx x r
rotateL _ _ _ Tip = error "rotateL Tip"
```

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

singleL

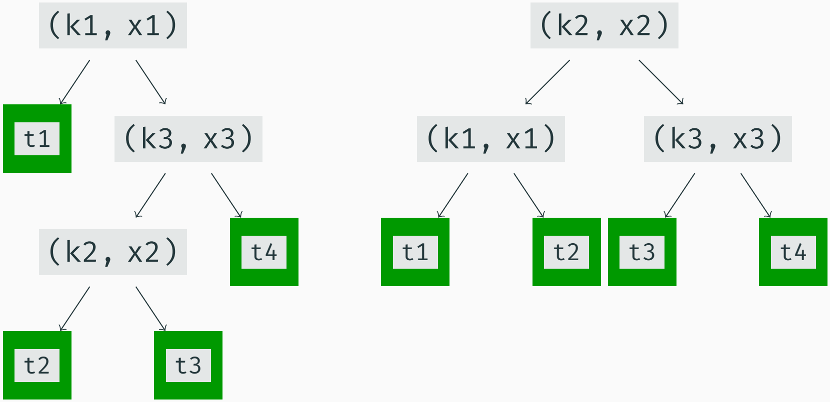
```haskell
singleL :: Map k a -> k -> a -> Map k a -> Map k a
singleL t1 k1 x1 (Bin _ t2 k2 x2 t3) =
  bin (bin t1 k1 x1 t2) k2 x2 t3
```





doubleL

```haskell
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 (cntd.)

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*.

Excursion: nested datatypes

Nested datatypes

A nested datatype is a recursive parameterized datatype where the recursive occurrences change the parameter.

Nested datatypes

A nested datatype is a recursive parameterized datatype where the recursive occurrences change the parameter.

Non-example:

```
data List a = Nil | Cons a (List a)
```

The recursive occurrence `List a` leaves parameter `a` unchanged.

Nested datatypes

A nested datatype is a recursive parameterized datatype where the recursive occurrences change the parameter.

Non-example:

```
data List a = Nil | Cons a (List a)
```

The recursive occurrence `List a` leaves parameter `a` unchanged.

Example:

```
data Perfect a = Zero a | Suc (Perfect (a, a))
```

The recursive occurrence `Perfect (a, a)` changes parameter `a` to `(a, a)`.

Shape invariants

- Nested datatypes allow expressing certain shape invariants of datatypes that can otherwise not easily be expressed.
- For example, as we shall see in a moment, `Perfect` is a datatype representing perfect trees, i.e., complete, perfectly balanced, binary trees.
- Similarly, `FingerTree` ensures that nodes are simple near the top and grow more complex the deeper you go.

Constructing perfect trees

```
data Perfect a = Zero a | Suc (Perfect (a, a))
```

Constructing perfect trees

```
data Perfect a = Zero a | Suc (Perfect (a, a))
```

```
zero :: Perfect Int
zero = Zero 1
```

Constructing perfect trees

```
data Perfect a = Zero a | Suc (Perfect (a, a))
```

```
zero :: Perfect Int
zero = Zero 1
```

```
one :: Perfect Int
one = Suc (Zero (1, 2))
```

Constructing perfect trees

```haskell
data Perfect a = Zero a | Suc (Perfect (a, a))
```

```haskell
zero :: Perfect Int
zero = Zero 1
```

```haskell
one :: Perfect Int
one = Suc (Zero (1, 2))
```

```haskell
two :: Perfect Int
two = Suc (Suc (Zero ((1, 2), (3, 4))))
```

Defining functions on nested datatypes

Defining functions on nested datatypes seems tricky:

```
sumPerfect :: Perfect Int -> Int
sumPerfect (Zero n) = n
sumPerfect (Suc p)  = ...
```

In order to call `sumPerfect` recursively on `p`, it would have to work on `Perfect (Int, Int)`!

Defining functions on nested datatypes

Defining functions on nested datatypes seems tricky:

```
sumPerfect :: Perfect Int -> Int
sumPerfect (Zero n) = n
sumPerfect (Suc p)  = ...
```

In order to call `sumPerfect` recursively on `p`, it would have to work on `Perfect (Int, Int)`!

But then, we'd also need a function that works on `Perfect ((Int, Int), (Int, Int))`.

Generalizing?

We could also try to define a more general function:

```
sumPerfect :: Perfect a -> Int
sumPerfect (Zero n) = ...
sumPerfect (Suc p)  = sumPerfect p
```

Now we don't know what to do in the first case, because we cannot turn an unknown `a` into `Int`.

Adding a continuation

But perhaps we can generalize over that, too ...

```haskell
sumPerfect' :: (a -> Int) -> Perfect a -> Int
sumPerfect' k (Zero n) = k n
sumPerfect' k (Suc p)  = sumPerfect' newk p
  where
    newk (a1, a2) = k a1 + k a2
```

Adding a continuation

But perhaps we can generalize over that, too ...

```
sumPerfect' :: (a -> Int) -> Perfect a -> Int
sumPerfect' k (Zero n) = k n
sumPerfect' k (Suc p)  = sumPerfect' newk p
  where
    newk (a1, a2) = k a1 + k a2
```

```
sumPerfect :: Perfect Int -> Int
sumPerfect p = sumPerfect' id p
```

Polymorphic recursion

```haskell
sumPerfect' k (Zero n) = k n
sumPerfect' k (Suc p)  = sumPerfect' newk p
  where
    newk (a1, a2) = k a1 + k a2
```

Omitting the type signature on `sumPerfect'` results in a type error:

Occurs check: cannot construct the infinite type: `t ~ (t, t)`
Expected type: `((t, t) -> p)) -> Perfect (t, t) -> p`
Actual type: `(t -> p) -> Perfect t -> p`

Polymorphic recursion (contd.)

- Functions which invoke themselves recursively at a different type are called polymorphically recursive.
- Haskell's (Damas-Hindley-Milner-based) type inference algorithm can check, yet not infer polymorphically recursive types, so type signatures for such functions are required.
- Functions on nested types are naturally polymorphically recursive.

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.