计算概论A—实验班 函数式程序设计 Functional Programming

胡振江,张伟

北京大学 计算机学院 2024年09~12月

第14章: Foldables and Friends

主要知识点: Monoid, Foldable, Traversal

Adapted from Graham's Lecture slides





与GHC的实现并不完全一致 米 我们按照GHC的实现进行讲解

米教材《Programming in Haskell》中关于Monoid的内容

Semigroup (半群)

Defined in Data.Semigroup

class Semigroup a where

The class of semigroups (types with an associative binary operation).

Instances should satisfy the following:

Associativity

Since: base-4.9.0.0

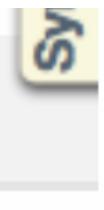
Minimal complete definition

(<>)

Methods

 $x \iff (y \iff z) = (x \iff y) \iff z$

(<>) :: a -> a -> a infixr 6







Monoid(幺半群) --Defined in Data.Monoid

Source

class Semigroup a => Monoid a where

The class of monoids (types with an associative binary operation that has an identity). Instances should satisfy the following:

Right identity

```
x <> mempty = x
```

Left identity

mempty <> x = x

Associativity

```
x \iff (y \iff z) = (x \iff y) \iff z (Semigroup law)
```

Concatenation

mconcat = foldr (<>) mempty

The method names refer to the monoid of lists under concatenation, but there are many other instances.

Some types can be viewed as a monoid in more than one way, e.g. both addition and multiplication on numbers. In such cases we often define newtypes and make those instances of Monoid, e.g. Sum and Product

NOTE: Semigroup is a superclass of Monoid since base-4.11.0.0.

Minimal complete definition

mempty

S	Methods						
nas	mempty :: a	# Sour					
	Identity of mappend						
	mappend :: a -> a -> a	# Sou					
ut	An associative operation NOTE : This method is redundant and has the default implementation mappend = (<>) since <i>base-4.11.0.0</i> . Should it be implemented manually, since mappend is a synonym for (<>), it is expected that two functions are defined the same way. In a future GHC release mappend will be removed from Monoid.						
ct.	mconcat :: [a] -> a	# Sou:					
	Fold a list using the monoid. For most types, the default definition for mconcat will the function is included in the class definition so that an	-					

version can be provided for specific types.



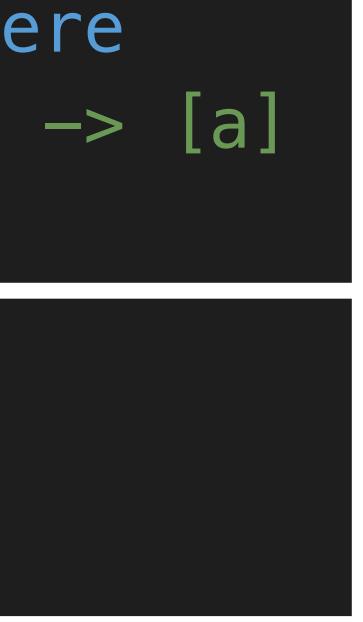


instance Semigroup [a] where -- (<>) :: [a] -> [a] -> [a] (<>) = (++)

instance Monoid [a] where -- mempty :: [a] mempty = []

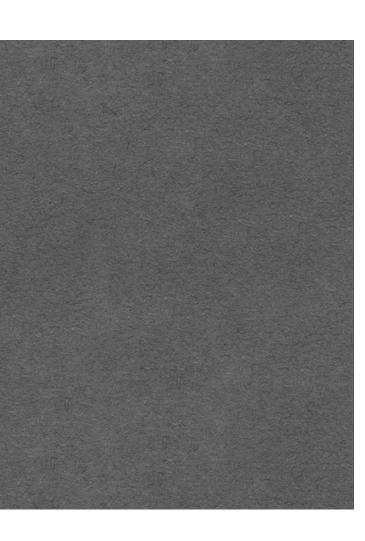
ghci> [1,2,3] <> [4,5,6] [1, 2, 3, 4, 5, 6]ghci> [1,2,3] <> mempty [1, 2, 3]

List Vonoic



Defined in Data.Semigroup

Defined in Data.Monoid





Naybe Monoid

instance Semigroup a => Semigroup (Maybe a) where --(<>) :: Maybe a -> Maybe a -> Maybe a Nothing <> b = ba <> Nothing = a Just a <> Just b = Just (a <> b)

instance Semigroup a => Monoid (Maybe a) where -- mempty :: Maybe a mempty = Nothing

Defined in Data.Semigroup

Defined in Data.Monoid





Int Monoid

A particular type may give rise to a monoid in a number of different ways.

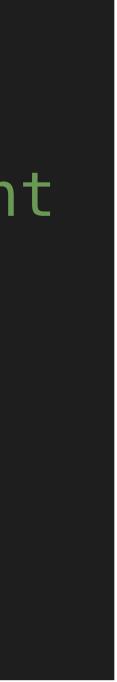
instance Semigroup Int where -- (<>) :: Int -> Int -> Int (<>) = (+)

instance Monoid Int where -- mempty :: Int mempty = 0

* But, multiple instance declarations of the same type for the same class are not permitted in Haskell!

instance Semigroup Int where -- (<>) :: Int -> Int -> Int (<>) = (*)

instance Monoid Int where -- mempty :: Int mempty = 1





Sum Monoid -- Defined in Data.Semigroup Data.Monoid

newtype Sum a = Sum aderiving (Eq, Ord, Show, Read)

getSum :: Sum a -> a getSum (Sum x) = x

instance Num a => Semigroup (Sum a) where -- (<>) :: Sum a -> Sum a -> Sum a -> Sum a Sum $x \ll Sum y = Sum (x + y)$

instance Num a => Monoid (Sum a) where -- mempty :: Sum a mempty = Sum 0

ghci> import Data.Monoid ghci> mconcat [Sum 2, Sum 3, Sum 4] Sum 9



Product Monoid -- Defined in Data.Semigroup Data.Monoid

newtype Product a = Product a deriving (Eq, Ord, Show, Read)

getProduct :: Product a -> a getProduct (Product x) = x

instance Num a => Semigroup (Product a) where -- (<>) :: Product a -> Product a -> Product a Product $x \ll Product y = Product (x * y)$

instance Num a => Monoid (Product a) where -- mempty :: Product a mempty = Product 1

Product 24

ghci> import Data.Monoid ghci> mconcat [Product 2, Product 3, Product 4]





Bool Monoid -- Defined in Data.Semigroup Data.Monoid

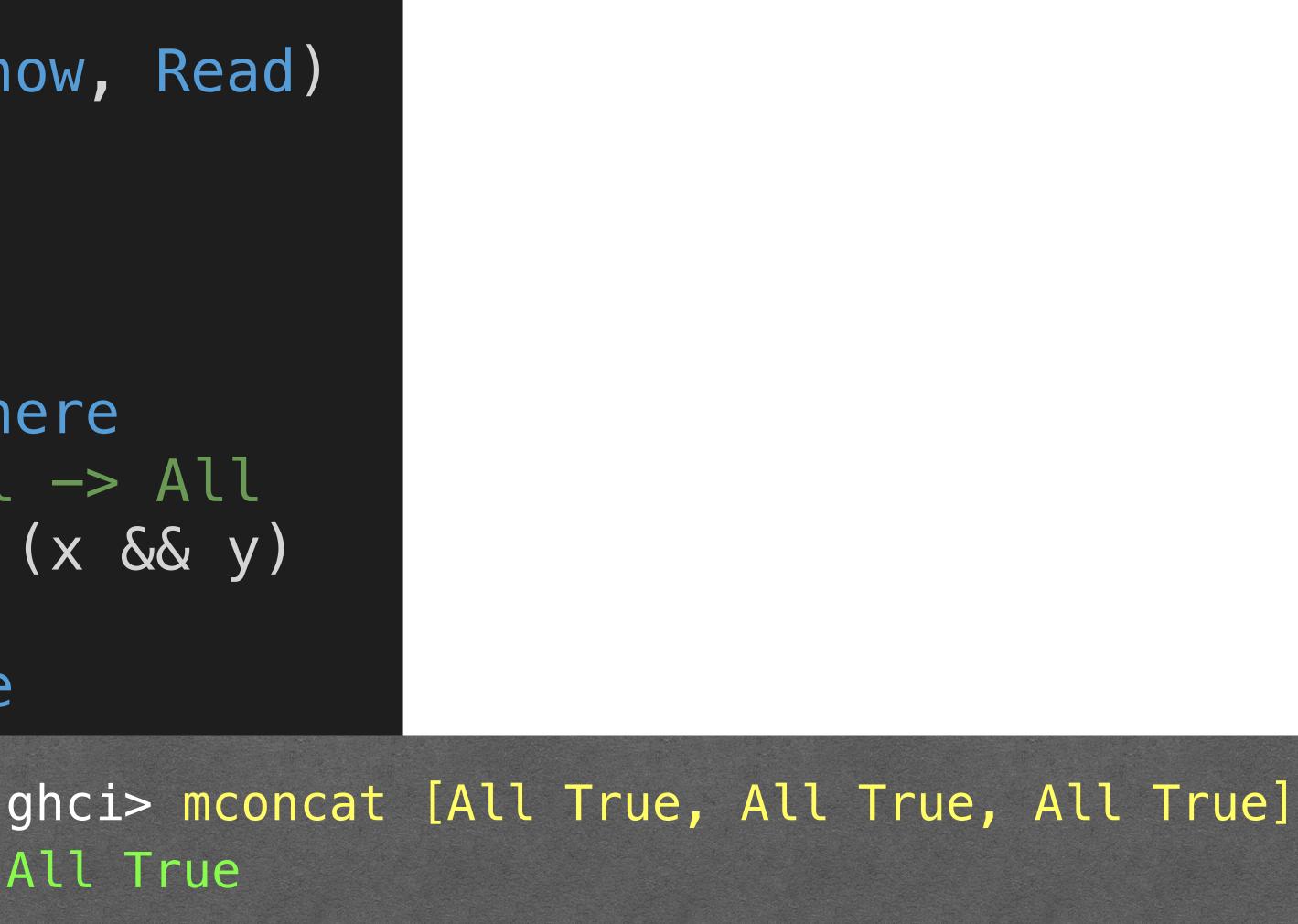
newtype All = All Boolderiving (Eq, Ord, Show, Read)

getAll :: All -> Bool getAll (All x) = x

instance Semigroup All where -- (<>) :: All -> All -> All All $x \ll All y = All (x \& y)$

instance Monoid All where -- mempty :: All mempty = All True

All True All False



ghci> mconcat [All True, All True, All False]



Bool Monoid -- Defined in Data.Semigroup Data.Monoid

newtype Any = Any Boolderiving (Eq, Ord, Show, Read)

getAny :: Any -> Bool getAny (Any x) = x

instance Semigroup Any where -- (<>) :: Any -> Any -> Any Any $x \ll Any y = Any (x | | y)$

instance Monoid Any where -- mempty :: Any mempty = Any False

ghci> mconcat [Any True, Any True, Any False] Any True ghci> mconcat [Any False, Any False, Any False] Any False





Foldable

combine all the values in a list to give a single value.

fold [] = mempty

Fold provides a simple means of "folding up" a list using a monoid:

fold :: Monoid a => [a] -> a fold (x:xs) = x <> fold xs







deriving Show

fold :: Monoid a => Tree a -> a fold (Leaf x) = x fold (Node l r) = fold l <> fold r

data Tree a = Leaf a | Node (Tree a) (Tree a)



Foldable Class -- Defined in Data.Foldable

class Foldable t where fold :: Monoid a => t a -> a foldMap :: Monoid b => (a -> b) -> t a -> b foldr :: (a -> b -> b) -> b -> t a -> b foldl :: (b -> a -> b) -> b -> t a -> b



instance Foldable -- Defined in Data.Foldable

instance Foldable [] where -- fold :: Monoid a => [a] -> afold [] = mempty fold(x:xs) = x <> fold xs

foldMap [] = mempty

foldr v [] = v

foldl v [] = v

- -- foldMap :: Monoid b => $(a \rightarrow b) \rightarrow [a] \rightarrow b$ foldMap f (x:xs) = f x <> foldMap f xs
- -- foldr :: (a -> b -> b) -> b -> [a] -> b
- foldr f v (x:xs) = x `f` (foldr f v xs)
- -- foldl :: (b -> a -> b) -> b -> [a] -> b
- foldl f v (x:xs) = foldl f (v `f` x) xs



instance Foldable Tree

data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Show

instance Foldable Tree where -- fold :: Monoid a => Tree a -> a fold (Leaf x) = xfold (Node l r) = fold l <> fold r

> -- foldMap :: Monoid b => $(a \rightarrow b) \rightarrow Tree a \rightarrow b$ foldMap f (Leaf x) = f x foldMap f (Node l r) = foldMap f l <> foldMap f r

foldr f v (Leaf x) = x \hat{f} v foldr f v (Node l r) = foldr f (foldr f v r) l

-- foldl :: (b -> a -> b) -> b -> Tree a -> b foldl f v (Leaf x) = v f x

```
-- foldr :: (a -> b -> b) -> b -> Tree a -> b
```

```
foldl f v (Node l r) = foldl f (foldl f v l) r
```

Other Primitives and Defaults in Foldable

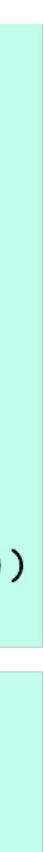
null	::	t a	->	Bool	L	
length	::	t a	->	${\tt Int}$		
elem	::	Eq a	a =>	a -	-> t	a ->
maximum	::	Ord	a =	> t	a ->	> a
minimum	::	Ord	a =	> t	a ->	> a
sum	::	Num	a =	> t	a ->	> a
product	::	Num	a =	> t	a ->	> a
foldr1 :	: (a ->	a –	> a)	->	t a
foldl1 :	: (a –>	a –	> a)	->	t a
toList :	: t	a –	> [a	a]		

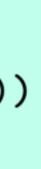
> Bool

-> a -> a

> null [] True
> null (Leaf 1) False
> length [110] 10
<pre>> length (Node (Leaf 'a') (Leaf 'b') 2</pre>
> foldr1 (+) [110] 55
> foldl1 (+) (Node (Leaf 1) (Leaf 2) 3







Foldable Class -- Defined in Data.Foldable

class Foldable t where fold :: Monoid a => t a -> a foldMap :: Monoid b => (a -> b) -> t a -> b foldr :: (a -> b -> b) -> b -> t a -> b foldl :: (b -> a -> b) -> b -> t a -> b



fold fold toL

- fold = foldMap id
- foldMap f = foldr (mappend . f) mempty
- toList = foldMap ($x \rightarrow [x]$)



-

Define Generic Functions using Foldable

average :: Foldable t => t Int -> Int average ns = sum ns `div` length ns

ghci> average [1..10] 5

ghci> average \$ Node (Leaf 1) (Leaf 3)



Define Generic Functions using Foldable

import Data.Monoid (Any(Any, getAny), All(All, getAll))

and :: Foldable t => t Bool -> Bool and = getAll foldMap All

or :: Foldable t => t Bool -> Bool or = getAny . foldMap Any

> ghci> and [True, False, True] False True

ghci> or \$ Node (Leaf True) (Leaf False)



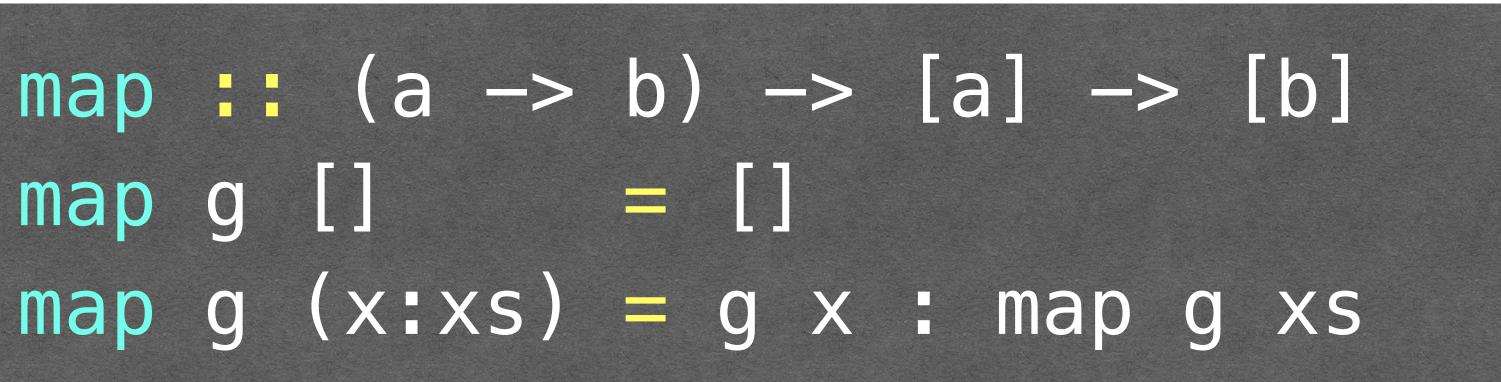


Motivation: generalizing map to deal with effects

map g [] = []

traverse :: (a -> Maybe b) -> [a] -> Maybe [b] traverse g [] = pure []





traverse g (x:xs) = pure (:) <*> g x <*> traverse g xs



traverse :: (a -> Maybe b) -> [a] -> Maybe [b] traverse g [] = pure [] traverse g (x:xs) = pure (:) <*> g x <*> traverse g xs

dec :: Int -> Maybe Int dec n = if n > 0 then Just (n-1)else Nothing

ghci> traverse dec [1,2,3] Just [0,1,2] ghci> traverse dec [2,3,0] Nothing

Traversa

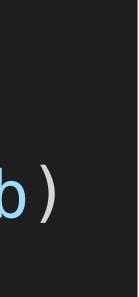


Taversable -- Defined in Data.Traversable

class (Functor t, Foldable t) => Traversable t where traverse :: Applicative f => (a -> f b) -> t a -> f (t b)

instance Traversable [] where traverse g [] = pure []

-- traverse :: Applicative f => (a -> f b) -> [a] -> f [b]traverse g (x:xs) = pure (:) <*> g x <*> traverse g xs



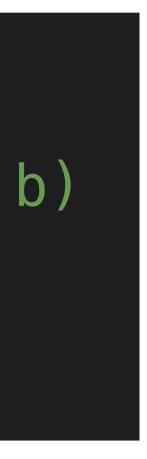


Traversable -- Defined in Data.Traversable

class (Functor t, Foldable t) => Traversable t where traverse :: Applicative f => (a -> f b) -> t a -> f (t b)

instance Traversable Tree where -- traverse :: Applicative f => (a -> f b) -> Tree a -> f (Tree b) traverse g (Leaf x) = Leaf <\$> g x traverse g (Node l r) = Node <\$> traverse g l <*> traverse g r





Other Primitives and Defaults in Traversable class (Functor t, Foldable t) => Traversable t where

traverse :: Applicative f => (a -> f b) -> t a -> f (t b)

In addition to the traverse primitive, the Traversable class also includes the following extra function and default definition:

sequenceA :: Applicative f => t (f a) -> f (t a) sequenceA =

> sequenceA [Just 1, Just 2, Just 3] Just [1,2,3]

> sequenceA [Just 1, Nothing, Just 3] Nothing

> sequenceA (Node (Leaf (Just 1)) (Leaf (Just 2))) Just (Node (Leaf 1) (Leaf 2))

> sequenceA (Node (Leaf (Just 1)) (Leaf Nothing)) Nothing





Other Primitives and Defaults in Traversable class (Functor t, Foldable t) => Traversable t where traverse :: Applicative f => (a -> f b) -> t a -> f (t b)

In addition to the traverse primitive, the Traversable class also includes the following extra function and default definition:

sequenceA :: Applicative f => t (f a) -> f (t a) sequenceA = traverse id

> sequenceA [Just 1, Just 2, Just 3] Just [1,2,3]

> sequenceA [Just 1, Nothing, Just 3] Nothing

> sequenceA (Node (Leaf (Just 1)) (Leaf (Just 2))) Just (Node (Leaf 1) (Leaf 2))

> sequenceA (Node (Leaf (Just 1)) (Leaf Nothing)) Nothing





Other Primitives and Defaults in Traversable class (Functor t, Foldable t) => Traversable t where traverse :: Applicative $f => (a \rightarrow f b) \rightarrow t a \rightarrow f (t b)$

Conversely, the class declaration also includes a default definition for traverse in terms of sequenceA, which expresses that to traverse a data structure using an effectful function we can first apply the function to each element using fmap, and then combine all the effects using sequenceA:

-- traverse :: Applicative f

traverse g =





Other Primitives and Defaults in Traversable class (Functor t, Foldable t) => Traversable t where traverse :: Applicative $f => (a \rightarrow f b) \rightarrow t a \rightarrow f (t b)$

Conversely, the class declaration also includes a default definition for traverse in terms of sequenceA, which expresses that to traverse a data structure using an effectful function we can first apply the function to each element using fmap, and then combine all the effects using sequenceA:

-- traverse :: Applicative f => $(a \rightarrow f b) \rightarrow t a \rightarrow f (t b)$ traverse g = sequenceA . fmap g







14-1 Show how the Maybe type can be made foldable and traversable, by giving explicit definitions for fold, foldMap, foldr, foldI and traverse.

14-2 In a similar manner, show how the following type of binary trees with data in their nodes can be made into a foldable and traversable type:

data Tree a = Leaf | Node (Tree a) a (Tree a) deriving Show

第14章: Foldables and Friends



Adapted from Graham's Lecture slides





