第十四章: Foldables and Friends

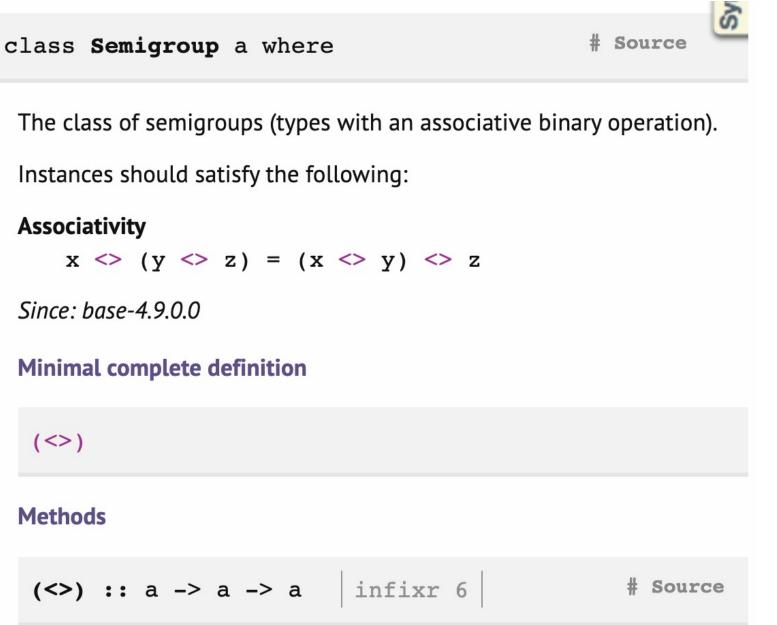
Monoids, Foldables, Traversals



- 教材《Programming in Haskell》中关于Moniods 的内容与GHC的实现并不完全一致
- 我们按照GHC的实现进行讲解



Defined in Data.Semigroup



3

Monoid (幺半群)

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 \iff mempty = x
```

Left identity

mempty <> x = x

Associativity

mempty

x <> (y <> z) = (x <> y) <> 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

Defined in Data.Monoid

Methodsmempty :: a# SourceIdentity of mappendmappend :: a -> a -> a# Source

An associative operation

S

Source

NOTE: This method is redundant and has the default implementation mappend = (<>) since *base-4.11.0.0*. Should it be implemented manually, since mappend is a synonym for (<>), it is expected that the two functions are defined the same way. In a future GHC release mappend will be removed from Monoid.

```
mconcat :: [a] -> a
```

Source

Fold a list using the monoid.

For most types, the default definition for mconcat will be used, but the function is included in the class definition so that an optimized version can be provided for specific types. List Monoid

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

Defined in Data.Semigroup

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

Defined in Data.Monoid

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

> [1,2,3] <> mempty
[1,2,3]

Maybe Monoid

instance Semigroup a => Semigroup (Maybe a) where --(<>) :: Maybe a -> Maybe a -> Maybe a Nothing <> b = b a <> 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 inData
.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
instance Semigroup Int where
   --- (<>) :: Int -> Int -> Int
   (<>) = (*)
instance Monoid Int where
   -- mempty :: Int
   mempty = 1
```

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

```
newtype Sum a = Sum a
      deriving (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 x \iff Sum y = Sum (x+y)
instance Num a => Monoid (Sum a) where
   -- mempty :: Sum a
   mempty = Sum 0
```

> mconcat [Sum 2, Sum 3, Sum 4]
Sum 9

```
newtype Product a = Product a
      deriving (Eq, Ord, Show, Read)
getProduct :: Product a -> a
qetProduct (Product x) = x
instance Num a => Semigroup (Product a) where
   -- (<>) :: Product a -> Product a -> Product a
   Product x <> Product y = Product (x * y)
instance Num a => Monoid (Product a) where
   -- mempty :: Sum a
  mempty = Product 1
```

> mconcat [Product 2, Product 3, Product 4]
Product 24

```
newtype All = All Bool
      deriving (Eq, Ord, Show, Read)
getAll :: All a -> a
getAll (All x) = x
instance Semigroup (All) where
   --- (<>) :: All -> All -> All
   All x \iff All y = All (x \& \& y)
instance Monoid (All) where
   -- mempty :: All
   mempty = All True
```

> mconcat [All True, All True, All True]
All True

```
newtype Any = Any Bool
      deriving (Eq, Ord, Show, Read)
getAny :: Any a -> a
qetAny (Any x) = x
instance Semigroup (Any) where
   --- (<>) :: Any -> Any -> Any
   Any x \iff Any y = Any (x \parallel y)
instance Monoid (Any) where
   -- mempty :: Any
   mempty = Any False
```

> mconcat [Any False, Any False, Any False]
Any False



Fold provides a simple means of "folding up" a list using a monoid: combine all the values in a list to give a single value.

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



Fold can also 'folding up' a tree using a monoid.

data Tree a = Leaf a | Node (Tree a) (Tree a)
 deriving Show
fold :: Monoid a => Tree a -> a

fold (Leaf x) = x
fold (Node l r) = fold l <> fold r



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 [] where

-- fold :: Monoid a => [a] -> a
fold [] = mempty
fold (x:xs) = x 'mappend' fold xs

-- foldMap :: Monoid b => (a -> b) -> [a] -> b
foldMap _ [] = mempty
foldMap f (x:xs) = f x 'mappend' foldMap f xs

-- foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ v [] = v
foldr f v (x:xs) = f x (foldr f v xs)

-- foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ v [] = v
foldl f v (x:xs) = foldl f (f v x) xs

> foldMap Sum [1..10]
Sum 55

> foldMap Product
[1..10]
Product 3628800



```
instance Foldable Tree where
   -- fold :: Monoid a => Tree a -> a
   fold (Leaf x) = x
   fold (Node l r) = fold l 'mappend' fold r
   -- foldMap :: Monoid b => (a \rightarrow b) \rightarrow Tree a -> b
   foldMap f (Leaf x) = f x
   foldMap f (Node l r) = foldMap f l 'mappend' foldMap f r
   -- foldr :: (a -> b -> b) -> b -> Tree a -> b
   foldr f v (Leaf x) = f x v
   foldr f v (Node l r) = foldr f (foldr f v r) l
   -- foldl :: (a -> b -> a) -> a -> Tree b -> a
   foldl f v (Leaf x) = f v x
   foldl f v (Node l r) = foldl f (foldl f v l) r
```

Other Primitives and Defaults

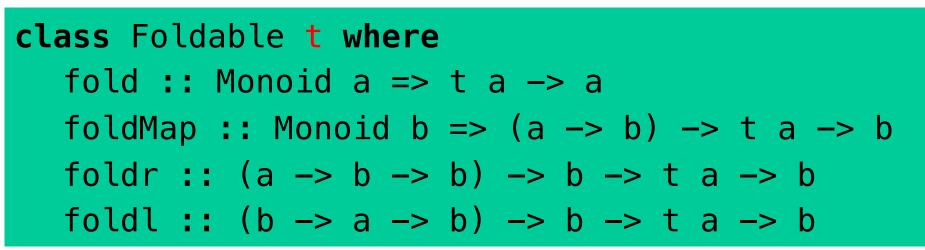
null	::	t a -> Bool
length	::	t a -> Int
elem	::	Eq a => a -> t a -> Bool
maximum	::	Ord a => t a -> a
minimum	::	Ord a => t a -> a
sum	::	Num a => t a -> a
product	::	Num a => t a -> a
foldr1 .	- ((a -> a -> a) -> t a -> a
	. ((a -> a -> a) -> t a -> a

toList :: t a -> [a]

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

Foldable Class

Defined in Data.Foldable



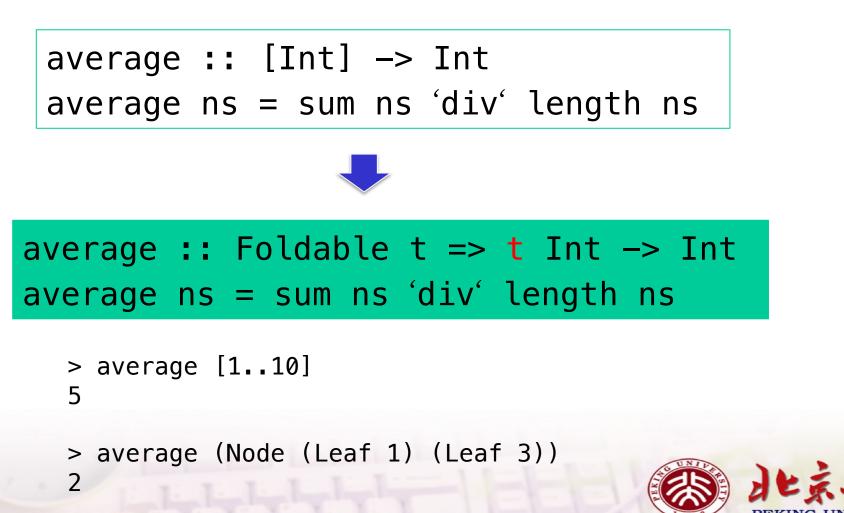
Minimal complete definition

foldMap		foldr	

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

Generic Functions

• The Foldable class helps us to define generic functions.



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

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

> and [True,False,True]
False

> or (Node (Leaf True) (Leaf False))
True



Traversals

• 动机: generalizing map to deal with effects

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



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



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

> traverse dec (Node (Leaf 1) (Leaf 2)) Just (Node (Leaf 0) (Leaf 1))

> traverse dec (Node (Leaf 0) (Leaf 1))
Nothing



Other Primitives and Defaults

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 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

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 -> f b) -> t a -> f (t b) traverse g = 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 -> f b) -> t a -> 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

