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

2025年09～12月

Adapted from Graham’s Lecture slides

第14章：Foldables and Friends

主要知识点：

Monoid、Foldable、Traversal

- 教材《Programming in Haskell》中关于Monoid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（但GHC似乎又有变化了）

Semigroup
(半群)

Defined in Data.Semigroup

```
class Semigroup a where
```

Source

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

Instances should satisfy the following:

Associativity

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

Since: base-4.9.0.0

Minimal complete definition

```
(<>)
```

Methods

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

Source

Monoid(幺半群) --Defined in Data.Monoid

```
class Semigroup a => Monoid a where        # Source
```

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 <> (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 `newtype`s 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`

Methods

```
mempty :: a        # Source
```

Identity of `mappend`

```
mappend :: a -> a -> a        # Source
```

An associative operation

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

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

Defined in Data.Semigroup

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

Defined in Data.Monoid

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

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

Maybe Monoid

```haskell
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)
```

Defined in Data.Semigroup

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

Defined in Data.Monoid

Int Monoid

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

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

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

```haskell
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!

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

```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 <> 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

```haskell
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 <> Product y = Product (x * y)

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

```
ghci> import Data.Monoid

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

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

```haskell
newtype All = All Bool
    deriving (Eq, Ord, Show, Read)

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

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

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

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

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

```haskell
newtype Any = Any Bool
    deriving (Eq, Ord, Show, Read)

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

instance Semigroup Any where
    -- (<>) :: Any -> Any -> Any
    Any x <> 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

- 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
```

Foldable

- Fold can also ‘folding up’ a tree using a monoid.

```haskell
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

```haskell
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

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

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

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

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

instance Foldable Tree

```haskell
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) = x
    fold (Node l r) = fold l <> fold r

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

    -- foldr :: (a -> b -> b) -> b -> Tree a -> b
    foldr f v (Leaf x) = x `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
    foldl f v (Node l r) = foldl f (foldl f v l) r
```

Other Primitives and Defaults in Foldable

```
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
foldl1 :: (a -> a -> a) -> t a -> a

toList :: t a -> [a]
```

```
> null []
True

> null (Leaf 1)
False

> length [1..10]
10

> length (Node (Leaf 'a') (Leaf 'b'))
2
```

```
> foldr1 (+) [1..10]
55

> foldl1 (+) (Node (Leaf 1) (Leaf 2))
3
```

Foldable Class -- Defined in Data.Foldable

```haskell
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
```

Minimal complete definition

`foldMap | foldr`

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

Define Generic Functions using Foldable

```haskell
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)
2
```

Define Generic Functions using Foldable

```haskell
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
ghci> or $ Node (Leaf True) (Leaf False)
True
```

Traversal

- Motivation: generalizing map to deal with effects

```haskell
map :: (a -> b) -> [a] -> [b]
map g []     = []
map g (x:xs) = g x : map g xs
```

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

Traversal

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

```haskell
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
```

Traversable -- Defined in Data.Traversable

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

```haskell
instance Traversable [] where
    -- traverse :: Applicative f => (a -> f b) -> [a] -> f [b]
    traverse g []     = pure []
    traverse g (x:xs) = pure (:) <*> g x <*> traverse g xs
```

Traversable -- Defined in Data.Traversable

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

```haskell
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

```haskell
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:

```haskell
sequenceA :: Applicative f => t (f a) -> f (t a)
sequenceA = [redacted]
```

```
> 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

```haskell
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:

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

```haskell
> 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

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

```haskell
-- traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
traverse g = [redacted]
```

Other Primitives and Defaults in Traversable

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

```haskell
-- traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
traverse g = sequenceA . fmap g
```

作业

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

就到这里吧