Week 8

Ch 12: Monoids

University of the Fraser Valley

Dr. Russell Campbell

Russell.Campbell@ufv.ca

COMP 481: Functional and Logic Programming

Overview

Programming Paradigms

There are two paradigms in programming that contrast each other, which you now know:

- procedural programming
  - uses functions and stores data within arrays
- object-oriented programming
  - arranges data as fields within a hierarchy of objects

Functional programming is very much like procedural programming, but to an extreme.

- arranging data into arrays allows for fast access to it during execution
- access to data within a hierarchy of objects can be slow, because of stepping through pointers to get to it

Arrays versus Records

Haskell has efficiency of arranging data apart from objects.

- the way we have used types up to this point follows more like object-oriented programming design
  - record syntax is not as efficient as another typing design we will learn

(Unity 3D is in the process of implementing an Entity Component System for Data Oriented Technology Stack to pack data into arrays)

— Wrapping an Existing Type Into a New Type —

Two Behaviours

We have seen in the last chapter the two ways that lists can implement the `<*>` operation:

1. with every nested function in the first list applied to every possible element of the second list

2. with `ZipList` each nested function of the first list applied to its corresponding element of the second list

The issue with `ZipList` is that it is implemented with the `data` keyword:

```
data ZipList a = ZipList { getZipList :: [a] }
```

newtype

For object-oriented-like design, Haskell wraps and unwraps the nested `a` and `[a]` types each time in use of `ZipList`.

- this is not as efficient as it could be

Haskell also has the `newtype` keyword for defining types so that Haskell treats the definition with its underlying type.

- the efficiency is tailored specifically to be used with implementations of functors and applicatives

- this is because the `newtype` keyword only allows exactly one constructor
  - also, the constructor can have up to only ***one*** field (in a record, if desired)

newtype with Functor Example (Simranjit Singh)

```
newtype Score a b =
    Score { getScore :: (String, b) }
        deriving Show

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

```
-- examples
p1 = Score ("James", 1)
p2 = Score ("Anna", 4)
p3 = Score ("Drew", 8)

ghci> getScore p1
("James", 1)

ghci> (+3) <$> p1
Score {getScore = ("James",4)}
```

— Using `newtype` to Make Type Class Instances —

Swapping Parameters

To implement functor on a constructor with two parameters so that functions `fmap` on its first parameter:

```
newtype Pair b a = Pair { getPair :: (a, b) } deriving (Show)

instance Functor (Pair b) where
    fmap g (Pair (x, y)) = Pair (g x, y)
```


- note the swap in the constructor for `Pair b a` and the record `getPair :: (a, b)`

Three Parameters

More parameters can be involved, and this time suppose we wanted to use the function on the second parameter:


```
newtype Triple a c b = Triple { getTriple :: (a, b, c) } deriving (Show)
```



```
instance Functor (Triple a c) where
    fmap g (Triple (x, y, z)) = Triple (x, g y, z)
```

Using `fmap`

Try `fmap` on a `Pair` and a `Triple` with some function. For example:

```
ghci> getPair (fmap reverse (Pair ("london calling", 3)))
("gnillac nodnol",3)
```


There is no change in the internal representation for Haskell with `Pair` or `Triple`.

- they only allow us to implement another way to use `fmap` (as an example) with pairs and 3-tuples.
- we could create yet more `newtype` and implementations for pairs and 3-tuples if we wanted

— On `newtype` and Laziness —

undefined

Haskell being lazy means it will only evaluate expressions until it absolutely must (e.g.: to print a result to output).

- only calculations that are necessary are performed, and no others
- trying to evaluate an `undefined` value (a special keyword) will make Haskell throw an exception


```
ghci> undefined
*** Exception: Prelude.undefined
CallStack (from HasCallStack):
  error, called at libraries\base\GHC\Err.hs:75:14 in base:GHC.Err
  undefined, called at <interactive>:56:1 in interactive:Ghci3
```

Ignoring `undefined`

however, notice how `undefined` can go unevaluated just fine when other calculations are the focus:


```
ghci> head [1,2,3,undefined,5,6]
1
```

`newtype` versus `data`

`newtype` allows structures to be more lazy than `data` because there is only exactly one constructor.

- Haskell must evaluate the parameters to determine which value constructor implementation matches with use of `data`
- the `newtype` value constructor implementation must only have the one version, so evaluation can be deferred


```
data CoolBool = CoolBool { getCoolBool :: Bool } deriving (Show)
```



```
helloMe :: CoolBool -> String
helloMe (CoolBool _) = "hello!"

helloMe undefined
*** Exception: Prelude.undefined
```

Demonstration of `newtype` Laziness

Exit interactive session and reenter to define the next version:

```
newtype CoolBool = CoolBool { getCoolBool :: Bool } deriving (Show)

helloMe :: CoolBool -> String
helloMe (CoolBool _) = "hello!"

ghci> helloMe undefined
"hello!"
```


- the `(CoolBool _)` did not need to evaluate `undefined`

Altogether, the difference between `data` and `newtype` is:

- make completely new types with `data`
- make other versions of types with `newtype`

— Type Keywords Review —

type

The `type` keyword is just used to create synonyms.

- e.g.: `type IntList = [Int]`
- it makes descriptions more readable like function signatures


```
ghci> ([1,2,3] :: IntList) ++ ([1,2,3] :: [Int])
[1,2,3,1,2,3]
```


- recall we created another name for the list association `[(String,String)]` as a `PhoneBook`

`newtype` Implementations

The `newtype` will create a completely separate type:

```
newtype CharList = CharList { getCharList :: [Char] } deriving
(Show)
```

- `CharList` values cannot have `++` applied with other `[Char]` values because the two types are different
- two `CharList` values cannot be concatenated with `++`, since `++` is not even implemented for `CharList`
  - it is possible to convert from `CharList` to `[Char]` apply `++` and convert back
  - `newtype` record syntax provides the field `getCharList` as conversion function
- none of the `[Char]` instance implementations are inherited to `CharList` for any of the involved type classes
  - you will need to derive or manually write them

Three Kinds of Data Implementation

Consider using `newtype` over `data` when you only have one value constructor.

The three canonical rules to follow are as follows:

1. If you only need **more readability**—the `type` synonym will do
2. If an already existing type needs to be wrapped and implemented as an **instance of a type class**—the `newtype` keyword will do
3. If a **completely new type** is needed—the `data` keyword will do

— Getting Back to Monoids —

Monoid Type Class

So we can implement instances of different type classes.

We have seen and learned of their usefulness:

- `Eq`, `Ord`, `Functor`, `Applicative`, etc.

There is yet another type class that is fairly involved and powerful, called `Monoid`:

- a `Monoid` describes a combination of
  - a binary function
  - with an identity value

Monoid Example

An example:

```
ghci> [1,2,3] ++ []
[1,2,3]
ghci> [] ++ [1,2,3]
[1,2,3]
```


- above, the binary function is `++`, and the identity element is `[]`
  - notice the identity element leaves the other operand unchanged, regardless of which side it is applied

Can you think of other examples?

`Monoid` implementation is as follows:

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```


- `mempty` describes the polymorphic constant that should act as the identity element
- `mappend` is a misnomer, and should be some associative binary operator
- `mconcat` is ***implemented by default*** to take a list and forms one monoid value using `mappend` between its elements
    - `mconcat` can have its default implementation changed depending on `m`

— The Monoid Laws —

Monoid Laws

The following three laws involving `Monoid` are not implemented in Haskell by default…

— left identity —

```
mempty `mappend` x = x
```


— right identity —


```
x `mappend` mempty = x
```


— associativity —

```
(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)
```

…so you will need to check your own implementations when you create instances!

Associativity

The last law requires a `Monoid` instance to ensure order of evaluation of `mappend` operations do not matter:

- once we implement last law, we can get away with writing

```
x `mappend` y `mappend` z
```


Part of the reason we want to guarantee such laws hold:

- we then do not have to change our understanding of a computational result based on order of operations

— A Few Monoids —

Implementing a Monoid Instance

We saw the example of `[]` and `++` function as a `Monoid` instance, which has the following implementation:

```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```


- the implementation of an instance of `Monoid` requires a concrete type, so note the use of `[a]` and not `[]`
- (requirement since version `build-4.11.0.0` for `Semigroup` to be a superclass of a `Monoid`, but we leave this for a bit later)

Examples of Using Monoid Behaviour

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

ghci> ("one" `mappend` "two") `mappend` "three"
"onetwothree"

ghci> "one" `mappend` ("two" `mappend` "three")
"onetwothree"

ghci> "one" `mappend` "two" `mappend` "three"
"onetwothree"

ghci> "ping" `mappend` mempty
"ping"

ghci> mconcat [[1,2],[3,6],[9]]
[1,2,3,6,9]

ghci> mempty :: [a]
[]
```

Details on Examples

(monoid behaviour)

- we needed a type annotation in the last expression, because Haskell would not be able to infer any type for it without giving `:: [a]`
  - it is more general to write `[a]` than `[Int]` or `[String]`, since lists could contain all its elements of any one type

- the expression `mconcat [[1,2],[3,6],[9]]` demonstrates how `++` is used between elements

The above examples demonstrated how `++` and `[]` satisfy the associativity and identity laws.

Commutativity

- there is no requirement to satisfy any commutativity laws
  - swapping order of elements in a `++` operation will be different
    - ghci> "tick" ++ "tock"
    - "ticktock“

    - ghci> "tock" ++ "tick"
    - "tocktick"

  - other monoids could be commutative over `mappend`:
    can you think of one?

  - to be commutative, the result of the binary operation must be the same, regardless of order,
    for **every** pair of possible operands

— Multiply and Sum —

Semigroup

Understand that both

- `+` and `0` is a monoid
- as well as `*` and `1` are a different monoid.

Recap: we can treat the same kind of thing with different implementations of type classes by repurposing them with `newtype`.

We want to see the different implementations for the two monoids, but the Haskell language has had an update since version `base-4.11.0.0`:

- any `Monoid` must also be a `Semigroup` instance
- `Semigroup` means that the associativity should be implemented and expected before implementing `Monoid`

We start with `Multiply`:

```haskell
newtype Multiply a = Multiply { getMultiply :: a }
    deriving (Eq, Ord, Read, Show, Bounded)


instance Num a => Semigroup (Multiply a) where
    (Multiply x) <> (Multiply y) = Multiply (x * y)
```



```haskell
instance (Num a) => Monoid (Multiply a) where
    mempty = Multiply 1
    mappend = (<>)
```


Now we can try the following operations using `<>`:

```
ghci> getMultiply $ Multiply 3 <> Multiply 9
ghci> getMultiply $ Multiply 3 <> mempty
ghci> getMultiply $ Multiply 3 <> Multiply 4 <> Multiply 2
ghci> getMultiply . mconcat . map Multiply $ [3,4,2]
```

Implement `Sum` as a Monoid Instance

Practice implementing `Sum` similarly to `Multiply`.

You will need to think about what the identity element is that corresponds to the operation of binary addition.

- once you get it implemented, test it with the following:


```
ghci> getSum $ Sum 2 <> Sum 9
11
ghci> getSum $ mempty <> Sum 3
3
ghci> getSum . mconcat . map Sum $ [1, 2, 3]
6
```

— Any and All —

`Bool` as a Monoid

We can work with `Bool` values and its common operations for our own monoids as well (like operators OR and AND).

- convince yourself of the `mempty` definition for the implementation where `<>` takes on the binary operation of OR:


```
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Ord, Read, Show, Bounded)

instance Semigroup (Any) where
    (Any x) <> (Any y) = Any (x || y)

instance Monoid Any where
    mempty = Any False
    mappend = (<>)
```

And give the following a try:

```
ghci> getAny $ Any True <> Any False
True
```



```
ghci> getAny $ mempty <> Any True
True
```



```
ghci> getAny . mconcat . map Any $ [False, False, False, True]
True

ghci> getAny $ mempty <> mempty
False
```

`Bool` as a Different Monoid

Can you implement another `<>` for the AND operator?

- it should have the following results:

```
ghci> getAll $ mempty <> All True
True
```



```
ghci> getAll $ mempty <> All False
False
```



```
ghci> getAll . mconcat . map All $ [True, True, True]
True

ghci> getAll . mconcat . map All $ [True, True, False]
False
```

— The Ordering Monoid —

Comparing Integers

The following slides are merely a demonstration of how to mix restrictions on comparing between `String` values.

- we have the three possible results of comparing integers:

```
ghci> 1 `compare` 2
LT
```



```
ghci> 2 `compare` 2
EQ
```



```
ghci> 3 `compare` 2
GT
```

- (could implement for `Char` as well)

Appending Ordering Values

Although it may not seem as if it would be possible to define `Monoid` behaviour over the three ordering values, it is already done so:


```
instance Monoid Ordering where
    mempty = EQ
    LT `mappend` _ = LT
    EQ `mappend` y = y
    GT `mappend` _ = GT
```

- you can check that the `mempty` acts like the identity on the other ordering values

Refined Comparison

The following might be a way we want to compare strings:

```
lengthCompare :: String -> String -> Ordering
lengthCompare x y =
    (length x `compare` length y) `mappend`
    (x `compare` y)
```


- the above will compare whether the first string `x` is shorter or larger and return that result

    - but if the two strings `x` and `y` are the same length, they will be further compared alphabetically


```
ghci> lengthCompare "zen" "ants"
LT
ghci> lengthCompare "zen" "ant"
GT
```

Even More Refined Comparison

We might want to design an a more refined scheme.

Further refine comparison to check between `x` and `y` by number of vowels before alphabetic comparison:


```
lengthCompare :: String -> String -> Ordering
lengthCompare x y =
    (length x `compare` length y) `mappend`
    (vowels x `compare` vowels y) `mappend`
    (x `compare` y)
 where vowels = length . filter (`elem` "aeiou")
```

Example of Orderings

See how the above refinement of ordering affects results with use of `lengthCompare`:

```
ghci> lengthCompare "zen" "anna"
LT
```



```
ghci> lengthCompare "zen" "ana"
LT
```



```
ghci> lengthCompare "zen" "ann"
GT
```

This is only one example of how to apply `Monoids` in a non-trivial way toward creating your own orderings.

— `Maybe` the Monoid —

Nested Monoids

Already implemented is the use of `Maybe` as a monoid (and a semigroup).

- the nested elements can also be instances of `Monoid` and `Semigroup`, so the implementations look like the following:


```haskell
instance Semigroup a => Semigroup (Maybe a) where
    (Just x) <> Nothing = Just x
    Nothing <> (Just y) = Just y
    (Just x) <> (Just y) = Just (x <> y)

instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    mappend = (<>)
```

- notice how `x <> y` nests in right evaluation of a `Just` element

Ignoring Nothing

Give the following a try:

```
ghci> Nothing <> Just "andy"
Just "andy"
ghci> Just LT <> Nothing
Just LT
ghci> Just (Sum 3) <> Just (Sum 4)
Just (Sum {getSum = 7})
```


- we can use `Maybe` values with nested values for types previously defined to work with `Monoid` on their own, such as `Sum`

The above implementation of `Maybe` is very useful:

- for working with binary computations in the nested elements when we do not care that some of the `Maybe` values are `Nothing`
- since they are absorbed during calculations as an identity

Only the First Element

And `Maybe` values with nested element ***not*** a monoid?

- grab the values without worrying about nested operations

Create `newtype First` as implementation of `Maybe`:

```
newtype First a = First { getFirst :: Maybe a }
    deriving (Eq, Ord, Read, Show)
```



```
instance Semigroup (First a) where
    First (Just x) <> _ = First (Just x)
    First Nothing <> x = x
```



```
instance Monoid (First a) where
    mempty = First Nothing
    mappend = (<>)
```

Using `First` as a Monoid

This way, we can work with the ***first*** element that is not `First Nothing` given some operations with `<>`:

```
ghci> getFirst $ First (Just 'a') <> First (Just 'b')
Just 'a'
```



```
ghci> getFirst $ First Nothing <> First (Just 'b')
Just 'b'
```



```
ghci> getFirst $ First (Just 'a') <> First Nothing
Just 'a'
```

```
ghci> getFirst . mconcat . map First $ [Nothing, Just 9, Just 10]
Just 9
```

Last

The `Data.Monoid` module already has a `Last` data type implemented that works similarly

It always evaluates to the rightmost non-`Nothing` value:


```
ghci> import Data.Monoid as M
ghci> getLast . mconcat . map Last $ [Nothing, Just 9, Just 10]
Just 10
```


- note that Haskell cn imply the package prefix for unique names

— Folding with Monoids —

The `foldr` function:

- used to have different version found in the `Data.Foldable` module
- allows us similar operations on other types that act similar to lists
- it is now just implemented into the `Prelude` default version
- the default only used to work on lists

Trees are analog to lists where we could practice folding:

- observe we can fold over anything deemed instance of `Foldable`


```
ghci> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Just a quick reminder demo:

```
ghci> foldr (*) 1 [1,2,3]
6
```

Folding with Monoids

In this example, the first parameter is a binary operation:

- the second parameter is the starting accumulator value
- the third parameter is the list we want to fold

A few more examples:


```
ghci> foldl (+) 2 (Just 9)
11
```



```
ghci> foldl (||) False (Just True)
True
```


When folding right, whatever *function* we pass must have:

- its first parameter as the next input list element
- its second parameter as the accumulator

Folding with Nothing

Something that is a bit weird, but works, because of monoid behaviour:

```
ghci> foldl (||) False (Nothing)
False

ghci> foldl (||) True (Nothing)
True

ghci> foldl (&&) False (Nothing)
False

ghci> foldl (&&) True (Nothing)
True
```


- the above I just remember as the fold as acting on the identity

Foldable Trees

We make another type an instance of `Foldable` with the tree data structure we had previously defined:


```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show)
```

foldMap

To make something foldable, we must implement a function called `foldMap`, which is described with the type:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```


- the above can be broken down into:
  - `(a -> m)` as a function for input nested element type `a` to an element that can be operated on as a monoid `m`
  - the data structure `t a` we would like to fold
  - the final result of the fold `m` for one monoid value

Implementing `foldMap` for `Tree`

We need to implement `foldMap` function for our trees:


```
instance Foldable Tree where
    foldMap g EmptyTree = mempty
    foldMap g (Node x l r) =
        foldMap g l <>
        g x         <>
        foldMap g r
```

- the above implements the fold as an in-order traversal
- a tree that is empty just evaluates to the monoid `mempty` value
    - this way, when the recursive `foldMap` reaches empty leaf nodes, they resolve as identity operations to the other parent nodes

A Test Tree

Testing trees out in an interactive ghci session is a bit much to type, but it is easiest to do in multiline syntax:

```
:{
testTree = Node 5
 (Node 3
 (Node 1 EmptyTree EmptyTree)
 (Node 6 EmptyTree EmptyTree)
 )
 (Node 9
 (Node 8 EmptyTree EmptyTree)
 (Node 10 EmptyTree EmptyTree)
 )
:}
```

(lines are blocked after `Node 5` with one space at the front)

Folding `testTree`

Now you should be able to check:

```
ghci> foldl (+) 0 testTree
42
ghci> foldl (*) 1 testTree
64800
```


- the first result is just the sum of all the nested node values together
- the second result is just the product of all of the nested node values together

Using Nested Monoids

But we want to see `foldMap` in action:

```
ghci> getAny $ foldMap (\x -> Any $ x == 3) testTree
True
```


- the operation between nodes in the tree are by `<>` and not some function you might be thinking on the nested elements themselves

- we converted nested values within each node to something that has monoid behaviour, i.e.: an `Any` value (we defined earlier)

- the result is whether some node in the tree contains the value `3`

Applying Custom Nested Monoids

If instead we wanted to get the sum of all the elements, you can guess we would next use the monoid we defined earlier for `Sum`:


```
ghci> getSum $ foldMap (\x -> (Sum x)) testTree
42
ghci> getMultiply $ foldMap (\x -> (Multiply x)) testTree
64800
```

More Nested Monoids

We would likely not implement basic operations as those already defined to work with the fold functions.

Keep in mind the flexibility with each of the further implementations of `Monoid` we had written:


```
ghci> getAny $ foldMap (\x -> Any $ x > 15) testTree
False
```


The above checks whether any of the nodes in the `testTree` have a value above `15`, which it does not.

Conversion to Lists

We can even convert our tree into a list:

```
ghci> foldMap (\x -> [x]) testTree
[1,3,6,5,8,9,10]
```


- note that the `Monoid` operation `<>` performed on lists concatenates them
- we could change the order of the traversal in the implementation of `foldMap` for our trees

— Monad —

Monad Example (David Semke)

```
import Data.Monoid

data Box a = Box a
    deriving (Show)

instance Functor Box where
    fmap f (Box x) = Box (f x)

instance Applicative Box where
    pure x = Box x
    (Box f) <*> x = fmap f x
```

```
instance Monad Box where
    return x = Box x
    Box x >>= f = f x

combineIntoBox
    :: (Monoid m) => m -> m -> Box m

combineIntoBox a b = Box (a `mappend` b)

resultChars =
    Box "day" >>= (combineIntoBox "good")

resultAny = Box (Any False)
    >>= combineIntoBox (Any True)
```

Thank You!

Questions?