Functional Programming

BONUS 2 :: Answers

1 Monoids

1. (a) There is a monoid called the *free monoid*. It has this name because it allows you to give any type a monoid instance for free. Can you think what it is?

   Solution: The list monoid ($\langle [a], +\!\!+, [\,] \rangle$) because any data type can be wrapped into a list and thus have this monoid. Freebie.

   (b) Prove that the *free monoid* is a monoid.

   Solution:
   To be a monoid $\langle [a], +\!\!+, [\,] \rangle$must uphold the monoid laws:

1.	Left identiy	$\varnothing \oplus x = x$
2.	Right identity	$x \oplus \varnothing = x$
3.	Associativity	$(x \oplus y) \oplus z = x \oplus (y \oplus z)$

   Proof:
   Using the definition of $(+\!\!+)$:

   $$\begin{aligned} &(+\!\!+) :: [a] \to [a] \to [a] \\ &[\,] +\!\!+ ys \quad\;\; = ys \\ &(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys) \end{aligned}$$

   Left identity:

   $$\begin{aligned} &\quad [\,] +\!\!+ ys \\ &=\!=\!= \quad \text{\{-(++) -\}} \\ &\quad ys \end{aligned}$$

   Right identity:
   Case: $[\,]$

   $$\begin{aligned} &\quad [\,] +\!\!+ [\,] \\ &=\!=\!= \quad \text{\{-definition of } +\!\!+ \text{ -\}} \\ &\quad [\,] \end{aligned}$$

   Case: $(x : xs)$

   $$\begin{aligned} &\quad (x : xs) +\!\!+ [\,] \\ &=\!=\!= \quad \text{\{-definition of } +\!\!+ \text{ -\}} \\ &\quad x : (xs +\!\!+ [\,]) \\ &=\!=\!= \quad \text{\{-induction hypothesis -\}} \\ &\quad (x : xs) \end{aligned}$$

   And for the associativity proof see week 2 answers, reverse proof, lemma 2.

2. A semi–group is a type equipped with an associative binary operation i.e. a Monoid without an identity element.
   Write a type class to represent a Semigroup.

   Solution:

   class *Semi s* **where**
   $\quad$ *binop* :: $s \to s \to s$

3. (Code) Define two Semigroup and Monoid instances for Booleans that can live in the same Haskell file together without Haskell getting upset.

Solution:

```
newtype BooleOR = BooleOR Bool

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

instance Monoid BooleOR where
  mempty = BooleOR False
  mappend (BooleOR x) (BooleOR y) = BooleOR (x || y)

newtype BooleAND = BooleAND Bool

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

instance Monoid BooleAND where
  mempty = BooleAND True
  mappend (BooleAND x) (BooleAND y) = BooleAND (x && y)
```

4. † (a) What is the difference between $(xs +\!\!+ ys) +\!\!+ zs$ and $xs +\!\!+ (ys +\!\!+ zs)$?

Solution: Since this is a monoid the result of these two expressions is the same. However, $(xs +\!\!+ ys) +\!\!+ zs$ is very inefficient. This expression is an O(n^2) operation.

The intuition of why it is inefficient is that the left most list (*xs*) gets unpacked for each $+\!\!+$. For example, in the expression $(((xs +\!\!+ ys) +\!\!+ zs) +\!\!+ ps)$ you can imagine each element of *xs* being prefixed to *ps*, then each element of $(xs +\!\!+ ys)$ being prefixed to zs, and then each element of $((xs +\!\!+ ys) +\!\!+ zs)$ being prefixed to *ps*. This leads to *xs* being unpacked multiple times, a redundant operation.

However, because Haskell is lazy the reason it is inefficient is not as straightforward as that. The latent inefficiency is caused by the way that Haskell manages its memory. Instead of Haskell unpacking *xs* many times it actually stores the fact that it needs to do this as a big unevaluated *thunk* structure. When prompted for an answer, Haskell is forced to wade through the *thunk* structure to get the result, slowing it down to an O(n^2) operation.

(b) Which way does Haskell bracket $xs +\!\!+ ys +\!\!+ zs$ by default?

Solution: $xs +\!\!+ (ys +\!\!+ zs)$. $(+\!\!+)$ is deliberately defined to be right associative in the Prelude, binding to the right element since it is more efficient.

5. † (a) (Code) Define the monoid instance for:

```
newtype Function a = Function (a → a)
```

Solution:

```
instance Monoid (Function a) where
  mempty                            = Function id
  mappend (Function x) (Function y) = Function (x · y)
```

† (b) Prove that this obeys the monoid laws.

Solution:
Left identity proof:

```
    mappend mempty (Function x)
=== {-def. mempty -}
    mappend (Function id) (Function x)
=== {-def. mappend -}
    (Function (id · x))
=== {-⟨a → b, (·), id⟩ is a monoid -}
    Function x
```

Right identity proof:

```
    mappend (Function x) mempty
=== {-def. mempty -}
    mappend (Function x) (Function id)
=== {-def. mappend -}
    Function (x · id)
=== {-⟨a → b, (·), id⟩ is a monoid -}
    Function x
```

Associativity identity proof:

```
    (Function x `mappend` Function y) `mappend` Function z
=== {-def. mappend -}
    Function (x · y) `mappend` Function z
=== {-def. mappend -}
    Function ((x · y) · z)
=== {-⟨a → b, (·), id⟩ is a monoid -}
    Function (x · (y · z))
=== {-def. mappend -}
    Function x `mappend` Function (y · z)
=== {-def. mappend -}
    Function x `mappend` (Function y `mappend` Function z)
```

6. Is it possible to create an instance of a monoid that does not uphold the monoid laws? If so, why, and give an example by defining the instance and saying which laws it breaks. If not, explain how invalid instances are restricted.

Solution:

Yes.

Type classes can only specify that a given type will perform the specified functions. Type classes do not come with any promises about upholding any laws or axioms. The onus is on the person writing the code to ensure that they define the classes properly. Defining an instance of a monoid as something that does not uphold the laws is unwise since the instance will behave unpredictably.

The following *Monoid* instance using the power operator is an example of a monoid instance that does not uphold the laws (In code we write *x*^*y* to represent x^y).

```
instance Monoid Integer where
  mempty  = 1
  mappend = (^)
```

As a bonus exercise, you should check to see which laws are violated.

† 7. A group $(G, *)$ is a set G together with a binary operation $* : G \to G \to G$ satisfying the following properties:

1. (Associativity) $\forall\, x, y, z \in G$, $(x * y) * z = x * (y * z)$
2. (Identity) $\exists\, e \in G$, such that $\forall\, x \in G$: $x * e = x = e * x$
3. (Inverse) $\forall\, x \in G$, $\exists\, x^{-1} \in G$, such that: $x * x^{-1} = e = x^{-1} * x$

(a) (Code) Define a type class for groups.

Solution:

```
class Group a where
   op       :: a → a → a
   identity :: a
   inverse  :: a → a
```

or if you think about the fact that *groups* are monoids with an inverse:

```
class Monoid a ⇒ Group' a where
   inverse' :: a → a
```

(b) (Code) Define a sensible instance of your type class that upholds properties of a group.

Solution:
Addition over the Integers:

```
instance Group Integer where
   op a b    = a + b
   identity  = 0
   inverse a = − a
```