Problem One (Eq Laws: A Good Instance)

In order for type classes to work properly in the Haskell ecosystem, they have to follow certain algebraic properties, which are called “laws” in the Haskell community. The first three Eq Laws are the following (these are just the axioms of an equivalence relation for math nerds—you know who you are!):

- Reflexivity: \( \forall x. x == x \)
- Symmetry: \( \forall x, y. x == y \leftrightarrow y == x \)
- Transitivity: \( \forall x, y, z. x == y \land y == z \rightarrow x == z \)

In this problem we will look at an example of a potential instance of Eq which satisfies these laws (a good instance); in the next we'll look at one which doesn’t (a bad instance).

Note: When you prove that an instance does satisfy the laws, you must do so by structural induction (remember this from CS 131?). You may assume that the types \( a \) and \( b \) DO satisfy the laws in such a proof (if necessary).

Consider the following instance of the Functor Eq:

```haskell
data Pair a b = P a b

instance (Eq a, Eq b) => Eq (Pair a b) where
  -- (==) :: Pair a b -> Pair a b -> Bool
  (P x y) == (P x' y') = x == x' && y == y'
```

(a) Prove that this satisfies Reflexivity.

Any instance of a `Pair` must be in the form \((P \times y)\) for some \(x\) and \(y\) of types \(a\) and \(b\), respectively. By the assumption that types \(a\) and \(b\) satisfy Reflexivity, we have

\[ x == x \land y == y \]

Therefore, by the definition of \(==\) on `Pairs`, we have

\[ (P \times y) == (P \times y) \]

(QED).
(b) Prove that this satisfies Symmetry.

Let \( x, x' \) and \( y, y' \) be arbitrary expressions of types \( a \) and \( b \), respectively. Then:

\[
(P \times y) == (P \times' y')
\]

\[
\leftrightarrow \quad x == x' \&\& y == y' \quad -- \text{by the definition of } == \text{ on Pairs}
\]

\[
\leftrightarrow \quad x' == x \&\& y' == y \quad -- \text{by the assumption that types } a \text{ and } b \text{ satisfy Symmetry}
\]

\[
\leftrightarrow \quad (P \times' y') == (P \times y) \quad -- \text{by the definition of } == \text{ on Pairs}
\]

(c) Prove that this satisfies Transitivity.

Let \( x, x', x'' \) and \( y, y', y'' \) be arbitrary expressions of types \( a \) and \( b \), respectively. Then:

\[
(P \times y) == (P \times' y') \&\& (P \times' y') == (P \times'' y'')
\]

\[
\rightarrow \quad x == x' \&\& y == y' \quad -- \text{by the definition of } == \text{ on Pairs}
\]

\[
\&\& x' == x'' \&\& y' == y''
\]

\[
\rightarrow \quad x == x' \&\& y == y' \quad -- \text{by the assumption that types } a \text{ and } b \text{ satisfy Transitivity}
\]

\[
\rightarrow \quad (P \times y) == (P \times'' y'') \quad -- \text{by the definition of } == \text{ on Pairs}
\]

**Problem Two (Eq Laws: A Bad Instance)**

Now consider the following instance of the Eq type class:

```
data BadPair a = B a a

instance Eq a => Eq (BadPair a) where
  -- (==) :: BadPair a -> BadPair a -> Bool
  (B x y) == (B x' y') = x == y' & & y == x'
```

Note: If you want to prove that this instance does NOT satisfy a law, it suffices to provide a counter-example, i.e., a concrete instance which does not satisfy the law.

You could, for example, consider instances of (Pair Integer Integer) when looking for counter-examples.
(a) Does this instance satisfy Reflexivity? Circle one:  Yes  No

Now prove it:

**Proof by contradiction:**

Suppose Reflexivity holds. Then \((B \ 2 \ 4) == (B \ 2 \ 4)\). But then, by the definition of \(==\) on BadPairs, we would have \(2 == 4 \&\& 4 == 2\), which contradicts the assumption that any instances of the type \(a\) satisfy Reflexivity. Hence Reflexivity does not hold for BadPairs.

(b) Does this instance satisfy Symmetry? Circle one:  Yes  No

Now prove it:

Let \(x, x'\) and \(y, y'\) be arbitrary expressions of types \(a\) and \(b\), respectively. Then:

\[
(B \times y) == (B \times' y')
\]

\[
\leftrightarrow x == y' \&\& y == x' \quad \text{-- by the definition of \(==\) on Pairs}
\]

\[
\leftrightarrow y' == x \&\& x' == y  \quad \text{-- by the assumption that types a and b satisfy Symmetry}
\]

\[
\leftrightarrow (B \times' y') == (B \times y) \quad \text{-- by the definition of \(==\) on Pairs}
\]

(c) Does this instance satisfy Transitivity? Circle one:  Yes  No

Now prove it:

**Proof by contradiction:**

Suppose Transitivity holds. Then, because Integers satisfy Reflexivity, we have

\[
2 == 2 \&\& 4 == 4
\]

\[
\rightarrow (B \ 2 \ 4) == (B \ 4 \ 2) \&\& (B \ 4 \ 2) == (B \ 2 \ 4) \quad \text{-- by the definition of \(==\) on BadPairs}
\]

\[
\rightarrow (B \ 2 \ 4) == (B \ 2 \ 4) \quad \text{-- by Transitivity}
\]

\[
\rightarrow 2 == 4 \&\& 4 == 2 \quad \text{-- by the definition of \(==\) on BadPairs}
\]

Obviously, this contradicts the assumption that Integers satisfy Reflexivity!
Problem Three (Functor Laws: A Good Instance)

The laws for Functors are as follows, which should hold for any instance T of the Functor type class; the instance declaration for T must define an implementation of fmap. In the following, f and g are the functions being mapped over an instance x of T.

First Law: \( \forall x. \text{fmap id } x = \text{id } x \) (where \( \text{id } = \lambda x \to x \))

Second Law: \( (\text{fmap } f \cdot \text{fmap } g) = \text{fmap } (f \cdot g) \)

An alternate form of the second law is perhaps easier to work with:

Second Law’ \( \forall x,f,g. (\text{fmap } f (\text{fmap } g x)) = \text{fmap } (f \cdot g) x \)

Now consider the following instance of the Functor type class:

```haskell
data Pair a = P a a

instance Functor Pair where
  -- fmap :: (a \to b) \to Pair a \to Pair b
  fmap f (P x y) = P (f x) (f y)
```

(a) Prove that this satisfies the First Law

Any instance of Pair must be in the form \((P \times y)\) for some \(x\) and \(y\). Therefore,

\[
\text{fmap id } (P \times y) = (P (\text{id } x) (\text{id } y)) -- \text{by def of fmap}
\]
\[
= (P \times y) -- \text{by def of id}
\]
\[
= \text{id } (P \times y)
\]

(b) Prove that this satisfies the Second Law (note that you will have to consider arbitrary functions \(f\) and \(g\) of the appropriate types— because of the universal quantifier on these you cannot simply pick two particular functions and show that the law works in those cases: what about all the other possible functions?).

Any instance of Pair must be in the form \((P \times y)\) for some \(x\) and \(y\). Let \(f\) and \(g\) be arbitrary functions of the appropriate types. Therefore,

\[
\text{fmap } f (\text{fmap } g (P \times y)) = \text{fmap } f (P (g x) (g y)) -- \text{by def of fmap}
\]
\[
= (P (f (g x)) (f (g y))) -- \text{by def of fmap}
\]
\[
= (P (((f \cdot g) x)) (((f \cdot g) y))) -- \text{by def of } (.)
\]
\[
= \text{fmap } (f \cdot g) (P \times y) -- \text{by def of fmap}
\]
Problem Four (Functor Laws: A Bad Instance)
Consider the following instance of the Functor type class:

```haskell
data BadList a = BNil | BCons a (BadList a)

instance Functor BadList where
    -- fmap :: (a -> b) -> BadList a -> BadList b
    fmap _ BNil = BNil
    fmap f (BCons x xs) = (BCons (f x) xs)
```

(a) Does this instance satisfy the First Law? Circle one: Yes No
Now prove it:
Proof by contradiction:

\[
\text{fmap id (BCons 2 (BCons 3 BNil))} = (\text{BCons (id 2) BNil}) -- \text{by def of fmap}
\]

\[
= (\text{BCons 2 BNil}) -- \text{by def of id}
\]

\[
/= (\text{BCons 2 (BCons 3 BNil)})
\]

\[
= \text{id (BCons 2 (BCons 3 BNil))}
\]

(b) Does this instance satisfy the Second Law? Circle one: Yes No
Now prove it:
Let \(f\) and \(g\) be arbitrary functions of the appropriate types. Any instance of \(\text{BadList}\) must either be in the form \(\text{BNil}\) or \((\text{BCons}\ x\ \text{xs})\) for some \(x\) and \(\text{xs}\) of the appropriate types. Therefore,

\[
\text{fmap f (fmap g BNil)} = \text{fmap f BNil} -- \text{by def of fmap}
\]

\[
= \text{BNil} -- \text{by def of fmap}
\]

\[
= \text{fmap (f.g) BNil} -- \text{by def of fmap}
\]

\[
\text{fmap f (fmap g (BCons x xs))} = \text{fmap f (BCons (g x) BNil)} -- \text{by def of fmap}
\]

\[
= (\text{BCons (f(g x)) BNil}) -- \text{by def of fmap}
\]

\[
= (\text{BCons ((f.g)x) BNil}) -- \text{by def of (.)}
\]

\[
= \text{fmap (f.g) (BCons x xs)} -- \text{by def of fmap}
\]