Quiz (Week 6)

Functors

Question 1

Which of the following type definitions admit law-abiding instances of `Functor`?

1. ✔ `Maybe`
2. ✗ `String`
3. ✔ `(->) a` for any `a`
4. ✔ `(,) a` for any `a`
5. ✔ `IO`
6. ✔ `[ ]`
7. ✔ `Gen`
8. ✔

   `Tree` where:

   ```
   data Tree a = Leaf | Branch a (Tree a) (Tree a)
   ```

Every one of these is a functor except for `String`, which is not a type constructor and therefore cannot be a `Functor`. `IO` is an abstract type that implements `Functor`, as is `Gen`. The others have the following implementations:

```
instance Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f (Just x) = Just (f x)
  fmap f Nothing  = Nothing

instance Functor ((->) x) where
  fmap :: (a -> b) -> (x -> a) -> (x -> b)
  fmap ab xa x = ab (xa x)

instance Functor ((,) x) where
  fmap :: (a -> b) -> (x, a) -> (x, b)
  fmap f (x, a) = (x, f x)

instance Functor [ ] where
  fmap :: (a -> b) -> [a] -> [b]
```

```
  fmap = map

instance Functor Tree where
  fmap :: (a -> b) -> Tree a -> Tree b
  fmap f Leaf = Leaf
  fmap f (Branch x l r) = Branch (f x) (fmap f l) (fmap f r)
```

Question 2

Here is a data type definition for a non-empty list in Haskell.

```
data NonEmptyList a = One a | Cons a (NonEmptyList a)
```

Which of the following are law-abiding `Functor` instances for `NonEmptyList`?

1. ✔

```
instance Functor NonEmptyList where
  fmap f (One x) = One (f x)
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

2. ✗

```
instance Functor NonEmptyList where
  fmap f (One x) = One (f x)
  fmap f (Cons x xs) = One (f x)
```

3. ✗

```
instance Functor NonEmptyList where
  fmap f (One x) = One (f x)
  fmap f (Cons x xs) = Cons (f x) (Cons (f x) (fmap f xs))
```

4. ✗

```
instance Functor NonEmptyList where
  fmap f (One x) = One (f x)
  fmap f (Cons x xs) = Cons (f (f x)) (fmap f xs)
```

Option 1 obeys the functor laws. Proof by induction of the first law (`fmap id xs = xs`): Base case, when `xs = One x` :

```
fmap id (One x) = One (id x)    -- Definition of fmap
                = One x         -- Definition of id
                = xs            -- as required.
```

Inductive case, assuming `xs = Cons x xs'`, with the inductive hypothesis that `fmap id xs' = xs'`:

```
fmap id (Cons x xs') = Cons (id x) (fmap id xs')  -- Definition of fmap
                     = Cons x (fmap id xs')       -- Definition of id
                     = Cons x xs'                 -- Inductive hypothesis
                     = xs                         -- as required.
```

The composition law (`fmap f (fmap g xs) = fmap (f . g) xs`) follows from parametricity.

Options 2 and 3 do not obey the first law, as `fmap id (Cons 3 (One 1))` does not equal `Cons 3 (One 1)`, and option 4 is not type correct as `f :: a -> b`, not `a -> a`.

Applicatives

Question 3

Which of the following type definitions are examples of `Applicative`?

1. ✔ `Maybe`
2. ✗ `String`
3. ✔ `(->) a` for any `a`
4. ✗ `(,) a` for any `a`
5. ✔ `IO`
6. ✔ `[ ]`
7. ✔ `Gen`

Once again, `String` is not a type constructor, so cannot be an `Applicative`. Furthermore `(,) a` does not admit a law-abiding applicative instance either, as we cannot implement `pure`:

```
instance Applicative ((,) x) where
  pure :: a -> (x, a)
  pure a = (??? , a)
```

Of the other options, `Gen` and `IO` are both applicatives, as are `Maybe`, `(->) a` and `[ ]` as follows:

```
instance Applicative Maybe where
  pure x = Just x
  Just f <*> Just x = Just (f x)
  _      <*> _      = Nothing

instance Applicative ((->) x) where
  pure a = \x -> a
  (xf <*> xa) x = xf x (xa x)

instance Applicative [ ] where
  pure a = [ a ]
  [] <*> ys = []
  (f:fs) <*> xs = map f xs ++ (fs <*> xs)
```

Question 4

Here is a data type definition for a non-empty list in Haskell.

```
data NonEmptyList a = One a | Cons a (NonEmptyList a)
```

Which of the following are law-abiding `Applicative` instances for `NonEmptyList`?

1. ✔

```
instance Applicative NonEmptyList where
  pure x = Cons x (pure x)
  (One f) <*> (One x) = One (f x)
  (One f) <*> (Cons x _) = One (f x)
  (Cons f _) <*> (One x) = One (f x)
  (Cons f fs) <*> (Cons x xs) = Cons (f x) (fs <*> xs)
```

2. ✗

```
instance Applicative NonEmptyList where
  pure x = One x
  (One f) <*> (One x) = One (f x)
  (One f) <*> (Cons x _) = One (f x)
  (Cons f _) <*> (One x) = One (f x)
  (Cons f fs) <*> (Cons x xs) = Cons (f x) (fs <*> xs)
```

3. ✔

```haskell
instance Applicative NonEmptyList where
  pure x = One x
  One f <*> xs = fmap f xs
  (Cons f fs) <*> xs = fmap f xs `append` (fs <*> xs)
    where
      append (One x) ys = Cons x ys
      append (Cons x xs) ys = Cons x (xs `append` ys)
```

4. ✗

```haskell
instance Applicative NonEmptyList where
  pure x = Cons x (pure x)
  One f <*> xs = fmap f xs
  (Cons f fs) <*> xs = fmap f xs `append` (fs <*> xs)
    where
      append (One x) ys = Cons x ys
      append (Cons x xs) ys = Cons x (xs `append` ys)
```

Option 3 is analogous to the same `Applicative` instance we knows from regular lists, so is a valid `Applicative` instance. Options 2 and 4 don't obey the first `Applicative` law, `pure id <*> v = v`. Option 1 is also a valid applicative instance, analogous to the `ZipList` instance available in the Haskell standard library.

Question 5

Suppose I wanted to write a function `pair`, which takes two `Applicative` data structures and combines them in a tuple, of type:

```haskell
pair :: (Applicative f) => f a -> f b -> f (a, b)
```

Select all correct implementations of `pair`.

1. ✗

```haskell
pair fa fb = pure fa <*> pure fb
```

2. ✗

```haskell
pair fa fb = pure (,) <*> pure fa <*> pure fb
```

3. ✔

```haskell
pair fa fb = pure (,) <*> fa <*> fb
```

4. ✔

```haskell
pair fa fb = fmap (,) fa <*> fb
```

5. ✗ There is no way to implement this function.

Options 1 and 2 are not type correct.

Options 3 and 4 are both correct, and equivalent, as `fmap f x = (pure f <*> x)`, if the `Applicative` and `Functor` instances are law-abiding.

Monads

Question 6

Which of the following type definitions are examples of `Monad`, or admit law-abiding `Monad` instances?

1. ✔ `Maybe`
2. ✗ `String`
3. ✔ `(->) a` for any `a`
4. ✗ `(,) a` for any `a`
5. ✔ `IO`
6. ✔ `[ ]`
7. ✔ `Gen`
8. ✗
   `Tree` where:

```haskell
data Tree a = Leaf | Branch a (Tree a) (Tree a)
```

Monad instances can be written for `Maybe`, `(->) a` and `[ ]`, and `IO` and `Gen` are abstract types that also implement `Monad`. The `Tree` type is not a (straightforward) instance of `Monad`, as we would have to somehow transform a `Tree (Tree a)` into a `Tree a` in a structure-preserving way, which I'm pretty sure is impossible.

```
instance Monad Maybe where
  Just x  >>= f = f x
  Nothing >>= f = Nothing

instance Monad ((->) x) where
  (xa >>= axb) = \x -> axb (xa x) x

instance Monad [ ] where
  xs >>= each = concatMap each xs
```

Question 7

We wish to write a function `s` of type `[m a] -> m [a]`, for any monad `m`. It will unpack each given value of type `m a` and collect their results into a list. Which of the following is a correct implementation of this function?

1. ✗

```
s :: Monad m => [m a] -> m [a]
s [] = []
s (a:as) = do
  x <- a
  xs <- s as
  pure (x : xs)
```

2. ✗

```
s :: Monad m => [m a] -> m [a]
s [] = return []
s (a:as) = do
  x <- a
  xs <- as
  pure (x : xs)
```

3. ✔

```
s :: Monad m => [m a] -> m [a]
s [] = return []
s (a:as) = do
  x <- a
  xs <- s as
  pure (x : xs)
```

4. ✗

```
s :: Monad m => [m a] -> m [a]
s [] = return []
s (a:as) = do
  a
  s as
  pure (a : as)
```

Only answer 3 is type correct.

Question 8

Now suppose we wish to write a function `m` of type, which applies a given function to each element of a list and collects the results:

```
m :: Monad m => (a -> m b) -> [a] -> m [b]
```

What is a correct implementation of `m`?

1. ✗

```
m :: Monad m => (a -> m b) -> [a] -> m [b]
m f [] = []
m f (x:xs) = do
  y <- f x
  ys <- m f xs
  return (y:ys)
```

2. ✗

```
m :: Monad m => (a -> m b) -> [a] -> m [b]
m f [] = []
m f (x:xs) = do
  y <- f x
  ys <- f xs
  return (y:ys)
```

3. ✔

```
m :: Monad m => (a -> m b) -> [a] -> m [b]
m f = s . map f
```

4. ✗

```
m :: Monad m => (a -> m b) -> [a] -> m [b]
m = s . map
```

Only answer 3 is type correct.

Submission is already closed for this quiz. You can click here to check your submission (if any).