COMP3400
Assignment 3 Written

Paul Vrbik

June 2, 2023

Either

The Functor and Applicative for the `Either` data-type is as follows:

```
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x)  = Left x

instance Applicative (Either e) where
    pure             =  Right
    Left  e  <*>  _  =  Left e
    Right f  <*>  r  =  fmap f r
```

Question 1. [4 MARKS]

Show `Either` satisfies the *second* functor law:

```
8 fmap (g . h) = fmap g . fmap h
```

```
fmap g . fmap h $ Left e
  = fmap g $ fmap h $ Left e      -- by composition rules
  = fmap g $ Left e               -- apply (3)
  = Left e                        -- apply (3)
  = fmap (g.h) (Left e)           -- unapply fmap (g.h) with (3)
```

Thus 8 for any `Left e`. 2 marks

```
fmap (g.h) (Right a)
  = Right (g.h $ a)               -- by (2)
  = Right (g $ h a)               -- composition rules
  = g <$> (Right $ h a)           -- by (2)
  = g <$> (h <$> Right a)         -- by (2)
  = g <$> (h <$>) $ Right a       -- unapply h <$>
  = (g <$>) $ (h <$>) $ Right a   -- unapply g <$>
  = (g <$>) . (h <$>) $ Right a   -- composition rules
  = fmap g . fmap h $ Right a
```

Thus 8 for any `Right a`. 2 marks

Thus 8 is true for any `Either e a`.

Question 2. [6 MARKS]

Show `Either` satisfies the *third* applicative law:

```
9  x <*> pure y = pure (\g -> g y) <*> x
```

```
(Left e) <*> pure y
  = (Left) e                          apply (7)
  = fmap (\g -> g y) (Left e)         unapply (3)
  = Right (\g -> g y) <*> (Left e)    unapply (7)
  = pure (\g -> g y) <*> (Left e)     apply (5)
```

Thus 9 for any `Left e`. 2 marks

```
(Right a) <*> pure y
  = fmap a (Right y)                   apply (5) and (6)
  = Right (a y)                        apply (2)
  = Right (\g -> g y $ a)              unapply g
  = fmap (\g -> g y $ a) (Right a)     apply (7)
  = Right (\g -> g y) <*> (Right a)    unapply (7)
  = pure (\g -> g y) <*> (Right a)     apply (5)
```

Thus 9 for any `Right a`. 4 marks

Thus 9 is true for any `Either e a`.

ZipWith

Recall the alternate definition for a list applicative given in tutorial 8.

```
1 instance Functor [] where
2     fmap _ []     = []
3     fmap g (x:xs) = g x : (fmap g xs)
4
5 instance Applicative [] where
6     pure f = repeat f
7     [] <*> _ = []
8     _ <*> [] = []
9     (f:fs) <*> (x:xs) = (f x) : (fs <*> xs)
```

When writing your proofs *use the line numbers given above* when justifying your steps.

Question 3. [7 MARKS]

Show your Applicative satisfies the *second* applicative law:

```
pure (g x) = pure g <*> pure x
```

Let $k \in \mathbb{N}$ and

```
    P(k) <=>
      take k $ pure (g x) = take k $ pure g <*> pure x            1 mark
```

We use the following definition of `take`

```
10 take 0 _  = []
11 take _ [] = []
12 take k (x:xs) = x : (take (k-1) xs)
```

and the following definition of `repeat`

```
13 repeat x = x : (repeat x)
```

Base:

```
take 0 $ pure (g x)
  = []                                          apply (10)
  = take 0 $ pure g <*> pure x                unapply (10)
```

and thus `P(0)`. 1 mark

Induction hypothesis: Assume `P(k)`. 1 mark

Induction: Note we do not apply or unapply (6) but rather just take `pure = repeat`.

```
take (k+1) $ repeat (g x)
  = take (k+1) $ (g x) : (repeat $ g x)                    apply (13)
  = (g x) : (take k $ repeat (g x))                        apply (12)
  = (g x) : (take k $ repeat g <*> repeat x)                    by IH
  = take (k+1) $ (g x):(repeat g <*> repeat x)           unapply (12)
  = take (k+1) $ g:(repeat g) <*> x:(repeat x)            unapply (9)
  = take (k+1) $ (repeat g) <*> (repeat x)               unapply (13)
```

thus `P(k+1)`. 3 marks (must use IH)

In conclusion `P(k)` for any $k \in \mathbb{N}$ by the PMI. 1 mark

Question 4. [8 MARKS]

Show your Applicative satisfies the *forth* applicative law:

```
xs <*> (ys <*> zs) = (pure (.) <*> xs <*> ys) <*> zs
```

Let

```
    P(xs) <=>
      xs <*> (ys <*> zs) = (pure (.) <*> xs <*> ys) <*> zs
```

for any `ys, zs :: [a]` 1 mark

Base:

```
(pure (.) <*> [] <*> ys) <*> zs
  = ((.) <$> [] <*>) <*> zs        by Corollary Lec 8B
  = [] <*> zs                      apply (2)
  = []                             apply (7)
  = [] <*> (ys <*> zs)             apply (7)
```

and thus `P([])`. 1 mark

Induction hypothesis: Assume `P(xs)`. 1 mark

Induction: Notice if `ys` or `zs` is empty then `P(xs)` is trivially true by (8). 1 mark

Otherwise,

```
(x:xs) <*> ((y:ys) <*> (z:zs))
  = (x:xs) <*> ((y $ z) : (ys <*> zs))                   apply (9)
  = (x $ y $ z) : (xs <*> (ys <*> zs))                   apply (9)
  = (x $ y $ z) : ((pure (.) <*> xs <*> ys) <*> zs)      by IH
  = ((.) x y $ z) : ((pure (.) <*> xs <*> ys) <*> zs)    by composition rules
  = ((.) x y $ z) : (((.) <$> xs <*> ys) <*> zs)         by Corollary Lec 8B
  = ((.) x y) : ((.) <$> xs <*> ys) <*> (z:zs)           unapply (9)
  = ((.) <$> (x:xs) <*> (y:ys)) <*> (z:zs)               unapply (3)
  = (pure (.) <*> (x:xs) <*> (y:ys)) <*> (z:zs)          by Corollary Lec 8B
```

and thus `P(x:xs)`. 3 marks (must use IH)

In conclusion `P(xs)` for any `xs::[a]` by the PMI. 1 mark