Project (Written)

Due: Friday May 28, 2021 11:59 PM (Australian Eastern Standard Time)

Assignment description

Note: CrowdMark does not like angled brackets.

Interpret `|$|` as the infix functor and `|*|` as the infix applicative in the questions.

Use the correct notation in your solutions however.

Submit your assignment

Help

After you have completed the assignment, please save, scan, or take photos of your work and upload your files to the questions below. Crowdmark accepts PDF, JPG, and PNG file formats.

Q1 (13 points)

Consider a function that takes the *average* of a list of numbers.

```
average :: (Fractional a) => [a] -> a
average = undefined
> average [1, 2, 3]
2.0
```

Even though it is mathematically meaningless, let your base case be `[]`.

1. [3 points] Define a *primitive recursive* version of average

   ```
   p_average :: (Fractional a) => [a] -> a
   ```

2. [3 points] Define a *tail recursive* helper function

   ```
   h_average :: (Fractional a) => a -> a -> [a] ->  a
   ```

   that finds the average of a list.

3. [1 point] Define `average` via `h_average`.

4. [2 point] Define an iteration invariant for `h_average` that proves correctn

Time left Hide
15:07:18

5. [3 points] Prove `h_average` satisfies your iteration invariant.

6. [1 point] Define two distinct quick-checks for `average` that *both* use `Arbitrary [Float]`.

Q2 (5 points)

We learned, to derive an instance of a monad for a parametrized type that the type also needs to be an instance of a Functor and Applicative.

However Haskell does not actually care in what order these class instances are defined and *therefore* a Functor and Applicative can be defined using the Monad instance.

Complete the missing parts of the following declaration for the State Transformer type that uses the `(>>=)` to define `(|*|)` and `(|$|)`.

```
type State = Int
newtype ST a = S (State -> (a,State))

apply :: ST a -> State -> (a, State)
apply (S st) x = st x

instance Functor ST where
  -- fmap :: (a -> b) -> ST a -> ST b
  fmap g st = do ...

instance Applicative ST where
  -- pure :: a -> ST a
  pure x = S (\s -> (x,s))

  -- (|*|) :: ST (a -> b) -> ST a -> ST b
  stf |*| stx = do ...

instance Monad ST where
  --  (>>=) :: ST a -> (a -> ST b) -> ST b
  st >>= f = S (\s -> let (x,s') = apply st s in apply (f x) s')
```

Q3 (10 points)

Note crowdmark does not like angled brackets so let the infix functor be given by `|$|` and the infix applicative be given by `|*|`

Consider the following pattern for `zipping` lists:

```
zip1 :: (a -> b) -> [a] -> [b]
```

Time left Hide
15:07:18

```
> zip1 succ [1,2,3]
[2,3,4]

zip2 :: (a -> b -> c) -> [a] -> [b] -> [c]
> zip2 (+) [1,2,3] [4,5,6]
[5,7,9]

zip3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
> zip3 (\x y z -> x + y + z) [1,2,3] [4,5,6] [7,8,9]
[12, 15, 18]
-- and so on
```

For lists of different lengths, *truncate* to the length of the shorter list.

```
zip2
> zip2 (+) [1] [1,2,3]
[2]
> zip2 (+) [1,2,3] [1]
[2]
```

Let us pretend Haskell has no default list Applicative (or Functor) and write one that does list *zips* instead.

```
> (\x y z -> x + y + z) |$| [1,2,3] |*| [4,5,6] |*| [7,8,9]
[12, 15, 18]
```

Note: If you want to check your definitions in the REPL you must define a new list type as we are unable to overwrite the default.

1. [2 points] Towards defining an Applicative instance, define the functor

```
instance Functor [] where
  -- fmap :: (a -> b) -> [a] -> [b]
  fmap g xs = ...
```

2. [3 points] Define the Applicative instance that zips lists.

```
instance Applicative [] where
  -- pure :: a -> [a]
  pure x = ...

  -- (|*|) :: [a -> b] -> [a] -> [b]
  gs |*| xs = ...
```

3. [5 points] Show your Applicative satisfies the *third* applicative law

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

Time left Hide
15:07:18

Q4 (8 points)

Suppose we have the following datatype for a tree with values in its nodes.

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

Further suppose we have a Functor instance for this tree that does the obvious thing: apply a function to the *value* in each node and leaf.

```
instance Functor Tree where
  -- fmap :: (a -> b) -> Tree a -> Tree b
  fmap f (Leaf x) = Leaf $ f x
  fmap f (Node lt x rt) = Node (fmap f lt) (f x) (fmap f rt)
```

[8 points] Using induction on trees, show this functor satisfies the *second* functor law.

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

Q5 (6 points)

Consider the higher-order function `unfold`

```
unfold p h t x
  | p x       = []
  | otherwise = h x : unfold p h t (t x)
```

1. [3 points] Define `map f` using `unfold`.

2. [3 points] Define `iterate f` using `unfold`.

Time left Hide
15:07:18