CSE 230 Second Midterm, Fall 2023

Ranjit Jhala

December 7th, 2023

NAME ________________________________

SID ________________________________

- You have **75 minutes** to complete this exam.
- Where limits are given, **write no more** than the amount specified.
- Avoid seeing anyone else's work or allowing yours to be seen.
- Do not communicate with anyone but an exam proctor.
- **Good luck!**

Prelude: Common Data Types

In multiple parts of this exam, we will use the following data types `Nullable` and `List` which are variants of `Maybe` and `[]` from the standard library (`Prelude`)

```
data Nullable a = Null | Val a
  deriving (Eq, Ord, Show)

data List a = Emp | Cons a (List a)
    deriving (Eq, Ord, Show)
```

Also recall the *funny* arrays from the *first* midterm which are *functions* from `Int` (indices) to the value at that index

```
data Arr v = MkArr (Int -> v)
```

and the various operations we (ok, you) defined on them

```
init :: v -> Arr v
init v = MkArr (\_ -> v)

get :: Int -> Arr v -> v
get idx (MkArr f) = f idx

set :: Int -> v -> Arr v -> Arr v
set idx val (MkArr f) = MkArr (\i -> if i == idx then val else f i)

arr :: Arr Int
arr = set 3 1000 (set 2 100 (set 1 10 (init 0)))
```

Part 1: Functors

Q1: fmap for Nullable [10 pts]

Fill in the blanks to complete the definition of `fmap` for `Nullable`

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

    fmap _ Null    = ________________________________

    fmap f (Val x) = ________________________________
```

such that when you are done, you get the following behavior

```
-- >>> fmap (+10) Null
-- Null

-- >>> fmap (+10) (Val 10)
-- Val 20
```

Q2: fmap for List [10 pts]

Fill in the blanks to complete the definition of `fmap` for `List`

```
instance Functor List where
    fmap :: (a -> b) -> List a -> List b
    fmap _ Emp          = Emp
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

such that when you are done, you get the following behavior

```
-- >>> fmap (+1) Emp
-- Emp

-- >>> fmap (+1) (Cons 0 (Cons 10 (Cons 100 (Cons 1000 Emp))))
-- Cons 1 (Cons 11 (Cons 101 (Cons 1001 Emp)))
```

Q3: fmap for Arr [15 pts]

Fill in the blanks to complete the definition of `fmap` for `Arr`

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

    fmap f (MkArr a) = ________________________________
```

such that when you are done, you get the following behavior

```
-- >>> (get 0 arr, get 1 arr, get 2 arr, get 3 arr)
-- (0,10,100,1000)

-- >>> let arr' = fmap (+1) arr
-- >>> (get 0 arr', get 1 arr', get 2 arr', get 3 arr')
-- (1,11,101,1001)
```

Part 2: Applicatives

The `fmap` operation works nicely for a *single* argument – i.e. applying a function over a *single* "container" – but not so nicely for functions taking *multiple* inputs.

For example, consider the following functions

```
plus2 :: Int -> Int -> Int
plus2 x1 x2 = x1 + x2

plus3 :: Int -> Int -> Int -> Int
plus3 x1 x2 x3 = x1 + x2 + x3

plus4 :: Int -> Int -> Int -> Int -> Int
plus4 x1 x2 x3 x4 = x1 + x2 + x3 + x4
```

Wouldn't it be cool if we could do something like

```
-- >>> fmap2 plus2 [1,2,3] [10,20,30]
-- [11,22,33]

-- >>> fmap3 plus3 [1,2,3] [10,20,30] [100,200,300]
-- [111,222,333]

-- >>> fmap4 plus4 [1,2] [10,20] [100,200] [1000,2000]
-- [1111,2222]
```

wouldn't it be *even cooler* if we could replace `fmap2`, `fmap3`, `fmap4` etc. with just a single with a *single* special function? Consider the *Applicative* typeclass which defines an (infix) operator `<*>` that does just that:

```
class App f where
    (<*>) :: f (a -> b) -> f a -> f b
```

At the end of this problem, we will be able to write the above as

```
-- >>> fmap plus2 [1,2,3] <*> [10,20,30]
-- [11,22,33]
-- >>> fmap plus3 [1,2,3] <*> [10,20,30] <*> [100,200,300]
-- [111,222,333]

-- >>> fmap plus4  [1,2] <*> [10,20] <*> [100,200] <*> [1000,2000]
-- [1111,2222]
```

HINT: The <*> operator is *left-associative* i.e. you should read

```
e1 <*> e2 <*> e3 <*> ... <*> en
```

as

```
((((e1 <*> e2) <*> e3) <*> ...) <*> en)
```

Q4. `App` instance for `Nullable` [10pts]

First, fill in the blanks to complete the `App` instance for `Nullable`

```
instance App Nullable where

  (<*>) :: Nullable (a -> b) -> Nullable a -> Nullable b

  Null  <*> _     = ______________________________

  _     <*> Null  = ______________________________

  Val f <*> Val v = ______________________________
```

so that when you are done we get the following behavior

```
-- >>> fmap plus2 (Val 10)  <*> (Val 100)
-- Val 110
-- >>> fmap plus2 Null      <*> (Val 100)
-- Null
-- >>> fmap plus2  (Val 10) <*> Null
-- Null

-- >>> plus3 `fmap` (Val 10) <*> (Val 100) <*> (Val 1000)
-- Val 1110
-- >>> plus3 `fmap` Null <*> (Val 100) <*> (Val 1000)
-- Null
-- >>> plus3 `fmap` (Val 10) <*> Null <*> (Val 1000)
-- Null
-- >>> plus3 `fmap` (Val 10) <*> (Val 100) <*> Null
-- Null
-- >>> plus4 `fmap` (Val 10) <*> (Val 100) <*> (Val 1000) <*> (Val 10000)
-- Val 11110
```

Q5. `App` instance for `[]` [10pts]

First, fill in the blanks to complete the `App` instance for `[]`

```
instance App [] where

    (<*>) :: [a -> b] -> [a] -> [b]

    []     <*> _      = ________________________________

    _      <*> []     = ________________________________

    (f:fs) <*> (x:xs) = ________________________________
```

So that when you are done, we get the following behavior

```
-- >>> plus2 `fmap` [1,2,3] <*> [10,20,30]
-- [11,22,33]
-- >>> plus3 `fmap` [1,2,3] <*> [10,20,30] <*> [100,200,300]
-- [111,222,333]
-- >>> plus4 `fmap` [1,2] <*> [10,20] <*> [100,200] <*> [1000,2000]
-- [1111,2222]
```

Q6. `App` instance for `Arr` [15pts]

Finally, fill in the blanks to complete the `App` instance for `Arr`

```
instance App Arr where

    (<*>) :: Arr (a -> b) -> Arr a -> Arr b

    MkArr f <*> MkArr v =  ________________________________
```

so that when you are done, we get the following behavior

```
-- >>> (get 0 arr, get 1 arr, get 2 arr, get 3 arr)
-- (0,10,100,1000)
-- >>> let arr'' = fmap plus3 arr <*> arr <*> arr
-- >>> (get 0 arr'', get 1 arr'', get 2 arr'', get 3 arr'')
-- (0,30,300,3000)
```

Part 3: The "programmable semicolon"

Haskell is often called a language with a "programmable semicolon". Recall that

```
do {x1 <- e1; x2 <- e2; ... ; xn <- en; e}
```

is the same as

```
e1 >>= (\x1 -> (e2 >>= \x2 -> ... en >>= (\xn -> e)))
```

So depending on the implementation of >>= (and return) code that uses `do`-blocks can have quite different behavior.

Consider the following `foo` function

```
foo :: (Monad m) => m a -> m b -> m (a, b)
foo ma mb = do
    a <- ma
    b <- mb
    return (a, b)
```

which may alternatively be written as

```
foo ma mb = do {a <- ma; b <- mb; return (a, b)}
```

or as

```
foo ma mb = ma >>= (\a -> mb >>= (\b -> return (a, b)))
```

Q7: `foo` with Maybe [10pts]

Fill in the blanks below to write down what the preceding expression will evaluate to:

```
>>> foo Nothing Nothing
```

____

```
>>> foo (Just 10) Nothing
```

____

```
>>> foo Nothing (Just "a")
```

____

```
>>> foo (Just 10) (Just "a")
```

____

Q8: `foo` with `[]` [15pts]

HINT: see `Monad` instance for `[]` in cheat sheet

Fill in the blanks below to write down what the preceding expression will evaluate to:

```
>>> foo [1,2] ["a", "b"]
```

____

Q9: `foo` with `State` [15pts]

Recall the `ST` for State-transformers with `Int` state defined in class as

```
type ST a = S.State Int a
```

and consider the following definitions

```
burp :: ST String
burp = do
    n <- S.get
    S.put (n * 10)
    return (show n)

q9 :: ST (String, String)
q9 = foo burp burp
```

Fill in the blanks below to write down what the preceding expression will evaluate to

```
-- S.evalState :: ST a -> Int -> a

>>> S.evalState (foo burp burp) 0
```

____

```
>>> S.evalState (foo burp burp) 1
```

____

Monad Instance "Cheatsheet"

```
-- Maybe
data Maybe a = Nothing | Just a

-- Monad instance for Maybe
instance Monad Maybe where
    return :: a -> Maybe a
    return x = Just x

    (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing >>= _ = Nothing
    Just x  >>= f = f x

-- Monad instance for []
instance Monad [] where
    return :: a -> [a]
    return x = [x]

    (>>=) :: [a] -> (a -> [b]) -> [b]
    []     >>= f = []
    (x:xs) >>= f = f x ++ (xs >>= f)

-- State
data State s a = MkST (s -> (s, a))

-- Monad instance for State
instance Monad (State s) where
    return :: a -> State s a
    return s = MkST (\s -> (s, a))

    (>>=) :: State s a -> (a -> State s b) -> State s b
    (MkST f) >>= g = MkST (\s -> let (s', a)    = f s in
                                 let (MkST f') = g a in
                                 f' s')
```