CS:3820 Fall 2023
Midterm #1 Review Problems

1 Functions

In answering these questions, you should rely on functional extensionality: two functions `f` and `g` are equal if, for arbitrary argument `x`, `f x = g x`.

The following questions refer to the reverse composition operation `pipe`, defined by:

```
pipe f g x = g (f x)
```

1. **Identity for `pipe`.**
   Suppose that `f :: a → b`; show that `pipe f id = f`, and that `pipe id f = f`
2. **Associativity of `pipe`.**
   Suppose that `f :: a → b, g :: b → c, h :: c → d`; show that `pipe f (pipe g h) = pipe (pipe f g) h`

A function `f :: T → T` is *idempotent* if `f ∘ f = f`, that is to say, if repeating the function has no more effect than just calling it once.

3. **An idempotent.**
   Give an idempotent function `f :: Bool → Bool` that is *not* the identity function. (That is to say: there must be some argument `x` such that `f x ≠ x`.)
4. **Mapping idempotents.**
   Mapping a function over a list is defined by:

   ```
   map :: (a → b) → [a] → [b]
   map f []       = []
   map f (x : xs) = f x : map f xs
   ```

   Suppose `f :: a → a` is idempotent. Prove, by structural induction, that `map f` is idempotent as well.
5. **Iterating idempotents.**
   We define the iteration of `f` by:

   ```
   iterate :: Int → (a → a) → (a → a)
   iterate 1 f = f
   iterate n f = iterate (n - 1) f ∘ f
   ```

   Suppose that `f :: a → a` is idempotent, and let `n` be any `Int` greater than or equal to 1. Prove, by induction on `n`, that `iterate n f = f`

2 Relations on Lists

6. **Equivalence relations.**
   A function `p :: a → a → Bool` is an equivalence relation iff:

   - It is *reflexive*: for all `x :: a`, `p x x = True`.
   - It is *symmetric*: for all `x`, `y :: a`, If `p x y = True`, then `p y x = True`.
   - It is *transitive*: for all `x`, `y`, `z :: a`, if `p x y = True` and `p y z = True`, then `p x z = True`

   Define

   ```
   eql :: (a → a → Bool) → [a] → [a] → Bool
   eql p [] []             = True
   eql p (x : xs) (y : ys) = p x y && eql p xs ys
   eql p _ _               = False
   ```

   Suppose that `p` is an equivalence relation. Prove that `eql p` is an equivalence relation as well.

3 Snoc Lists

The following questions refer to reverse cons-lists, frequently called snoc lists. They are defined by:

```
data Tsil a = Lin | Snoc (Tsil a) a
```

We will also rely on a mapping function for snoc-lists, defined by:

```
pam :: (a → b) → Tsil a → Tsil b
pam Lin = Lin
pam (Snoc sx x) = Snoc (pam f sx) (f x)
```

7. **Snoc-lists are functors.**
   (a) Prove, by structural induction, that `pam id = id`.
   (b) Suppose that `f :: b → c` and `g :: a → b`. Prove, by structural induction, that `pam f ∘ pam g = pam (f ∘ g)`.
8. **Appending snoc-lists.**
   (a) (10 points) Write a function `ppa` which appends its second argument to its first.

   ```
   ppa :: Tsil a → Tsil a → Tsil a
   ```

   (b) (10 points) Prove, by structural induction, that `pam f` distributes over your implementation of `ppa`. That is to say, prove that for arbitrary `f`, `sx`, `sy`, `pam f (ppa sx sy = ppa (pam f sx) (pam f sy)`

4 Trees

The following questions concern the type of binary trees over `a`, given by:

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

9. **Mapping over trees.**
   Define a mapping function for trees, with the following type:

   ```
   mapt :: (a → b) → Tree a → Tree b
   ```

10. **Trees as functors.**
    Prove, by structural induction, that your `mapt` function satisfies the functor laws:
    (a) `mapt id = id`
    (b) `mapt f ∘ mapt g = mapt (f ∘ g)`
11. **Other map-like functions.**
    Here is an alternative definition for tree map:

    ```
    mapt' :: (a → b) → Tree a → Tree b
    mapt' f Leaf           = Leaf
    mapt' f (Branch l x r) = Branch (mapt' f r) (f x) (mapt' f l)
    ```

    Show that this cannot be the functor mapping operation over `Tree`, by showing that it violates one of the functor laws.

5 Maybe

The following questions concern the `Maybe` type, which is defined by

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

12. `Maybe` **as a monoid (i).**
    One possible set of instances for the `Maybe` type are:

    ```
    instance Semigroup (Maybe a) where
        Just a ◇ mb  = Just a
        Nothing ◇ mb = mb

    instance Monoid (Maybe a) where
        mempty = Nothing
    ```

    (a) (10 points) Prove that these instance satisfy the semigroup laws: that for arbitrary values `ma`, `mb`, `mc` of type `Maybe a`, `ma ◇ (mb ◇ mc) = (ma ◇ mb) ◇ mc`. You do *not* need induction in this proof.

    (b) (10 points) Prove that these instances satisfy the monoid laws: that for arbitrary value `ma` of type `Maybe a`, `mempty ◇ ma = ma = ma ◇ mempty`. You do *not* need induction in this proof.

13. `Maybe` **as a monoid (ii).**
    Alternatively, we could also have the following instance:

    ```
    instance Semigroup (Maybe a) where
        Nothing ◇ _ = Nothing
        _ ◇ Nothing = Nothing
        ma ◇ _      = ma
    ```

    Argue that this semigroup instance *cannot* extend to a monoid instance; that is, that there can be no value `e` of type `Maybe a` such that for any other value `ma`, `e ◇ ma = ma = ma ◇ e`