CTFP Ch7 Solutions

Q1

> Can we turn the Maybe type constructor into a functor by defining:
> `fmap _ _ = Nothing`
> which ignores both of its arguments?

Recollecting the Functor laws:

1. Preserve Identity: `fmap id = id`

```
(a) fmap id Nothing = id Nothing
(b) fmap id (Just x) = Just x
```

2. Preserve Composition: `fmap (g.f) = (fmap g . fmap f)`

```
fmap (g.f) (Just x) = (fmap g . fmap f) (Just x)
fmap (g.f) (Nothing) = (fmap g . fmap f) (Nothing)
```

Now, if we try applying equational reasoning for the above laws and **Just**:

```
// {xyz} translates to {applying defn of xyz} henceforth.
fmap id (Just x)
=> {fmap} Nothing
=> {id} id Nothing != id (Just x)

fmap (g.f) (Just x)
=> {fmap} Nothing
=> {fmap} fmap g Nothing
=> {fmap} fmap g (fmap f (Just x))
=> {composition} (fmap g . fmap f) (Just x)
```

So, while the provided definition meets the requirements of composing morphisms, it fails to meet the requirement of preserving identity.

Q2

> Prove functor laws for the reader functor. Hint: it's really simple.

Using the fmap defn for reader functor:

```
instance Functor ((->) r) where
    fmap f g = f . g
```

Applying equational reasoning with the functor laws:

```
fmap id f
=> {fmap} id . f
=> {composition} id f

fmap (g.f) h
=> {fmap} g.f.h
=> {composition being associative} g . (f.h)
=> {fmap on f.h} g . (fmap f h)
=> {fmap on entire expression} fmap g (fmap f h)
=> {composition} (fmap g . fmap f) h
```

Q3

> Implement the reader functor in your second favorite language (the first being Haskell, of course).

The question essentially requires the implementation of composition in C++.

```
template <typename A, typename B, typename R>
std::function<B(R)> fmap(std::function<A(R)> f, std::function<B(A)> g) {
  return [&](R input) {
    return g(f(input));
  };
}
```

Q4

> Prove the functor laws for the list functor. Assume that the laws are true for the tail part of the list you're applying it to (in other words, use induction).

Referred to this resource.

Using the fmap defn for the list functor:

```
instance Functor List where
    fmap _ Nil = Nil
    fmap f (Cons x t) = Cons (f x) (fmap f t)
```

Applying equational reasoning with the functor laws:

```
// Proving identity requirement
fmap id Nil
=> {fmap} Nil
=> {identity} id Nil

fmap id (Cons x t)
```

```
=> {fmap} Cons (id x) (fmap id t)
=> {identity} Cons x (fmap id t)
=> {induction} Cons x t
=> {identity} id (Cons x t)

// Proving composition requirement
fmap (g.f) Nil
=> {fmap} Nil
=> {fmap} fmap g Nil
=> {fmap} fmap g (fmap f Nil)
=> {composition} (fmap g . fmap f) Nil

fmap (g.f) (Cons x t)
=> {fmap} Cons (g.f x) (fmap g.f t)
=> {induction} Cons (g.f x) ((fmap g . fmap f) t)
=> {composition} Cons (g (f x)) ((fmap g) (fmap f t))
=> {fmap} fmap g Cons (f x) (fmap f t)
=> {fmap + composition} (fmap g) . (fmap f) (Cons x t)
```