Typeclasses and Laziness

Functional Programming with Haskell: Homework 3

Exercises

Recall that the exercises are divided into three parts. The "Important" ones are ones you really should do to understand the material; the "Highly Recommended" ones are a really good idea, but you'll probably be alright if you don't do them; and finally the "Good Practice" exercises are good for making sure you have a good grasp on this aspect of Haskell.

Important

Laziness

Laziness is a very fundamental part of Haskell, as everything in Haskell is lazy by default. There are ways of turning that off in specific places, and sometimes Haskell programmers will want to do that (for optimization, efficiency, etc); however, that's a bit more advanced than we're covering for the time being, so remember: everything in Haskell is lazy. (I think Haskell has it right; being lazy is so much less effort.)

Let's work a little bit more with laziness to make sure you're comfortable with its implication. First of all, make sure you understand and can explain the following code snippet that we saw in the slides:

```
fib n = fibs !! n
  where
    fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Play around with this until you are sure you could explain it to someone else.

Next, work a little bit on your own infinite-stream-of-numbers example. Using an infinite stream (an infinite list), implement the Sieve of Eratosthenes, which is an old algorithm for finding prime numbers. Namely, define the following value:

```
-- An infinite list of all prime numbers
primes = ...
```

Do this by starting with the infinite list `[1..]` of all numbers, and then filter out composite numbers using the `filter` function and the `mod` function. If you haven't encountered these before, this is a good time to learn to use Hoogle, the Haskell library search engine. (Just Google for "hoogle", and you'll find it.)

Typeclasses

Haskell has the following typeclasses built-in. The Haskell standard library defines these in more or less the same way, though I've simplified some of them for the sake of demonstration. We're using slightly different names and the `my` prefix so that you don't get name collisions with the standard library; if you'd like to see the real typeclasses, look up `Eq`, `Show`, `Num`, and `Ord` on Hoogle.

```haskell
-- Print a data type as a string. This is similar to Python's repr(),
-- and is meant to write things in a manner such that they can
-- be used as code. For instance, escape characters in strings
-- are written with backslashes, not printed.
class MyShow a where
  myShow :: a -> String

-- A numeric type. A few other numeric types exist
--to represent things that can be divided or floating point numbers.
class MyNum a where
  myPlus :: a -> a -> a
  myTimes :: a -> a -> a
  myNegate :: a -> a
  myAbs :: a -> a
  myFromInteger :: Integer -> a

-- Compare two elements and output whether they are equal.
class MyEq a where
  myEq :: a -> a -> Bool

-- Compare two elements of the type.
-- Output an ordering, which indicates whether the two are equal
-- or whether the first is greater than or less than the second.
-- Since this implies an ability to test for equality, this requires
-- the Eq typeclass as well.
data MyOrdering = Equal | Greater | Less
class Eq a => Ord a where
  myCompare :: a -> a -> MyOrdering
```

First of all, run the code above. Afterwards, implement the following instances; your task is to supply the implementation of each of the function.

```haskell
-- `Maybe a` represents a nullable value of `a`.
-- For example, a `Maybe Int` is a nullable int.
-- You must pattern match to check if it's null, or extract the actual value.
-- This is already defined for you in the standard library.
data Maybe a = Just a | Nothing

-- Display Maybe values, if the inner value is displayable.
instance MyShow a => MyShow (Maybe a) where
```

```
  ...

-- Implement arithmetic on nullable integers.
instance MyNum (Maybe Int) where
  ...
```

Polymorphic Functions

Next, let's implement some functions that use these typeclasses. You've already written functions to display and compare lists, so let's change those to work on *any* type that supports printing and comparison. Note that these functions already exist in the standard library, so if you're not using IHaskell, you may need to prefix their names with `my` to avoid name collisions. Thus, implement the following functions and typeclasses:

```
-- Find an element in a list.
-- Note the way we write constraints before the function, using a '=>'.
-- This function returns Nothing if the element doesn't exist in the list,
-- or the integer index of the element in the list.
find :: MyEq a => [a] -> a -> Maybe Int

-- Check if an element is in a list.
-- You may want to just use your `find` function to implement this.
elem :: MyEq a => [a] -> a -> Bool
```

Highly Recommended

Typeclasses

Using the typeclasses defined earlier, implement the following instances:

```
-- Implement equality checking for nullable values.
instance MyEq a => MyEq (Maybe a) where
  ...

-- Convert a list to a string, assuming all
-- its elements can be converted to strings.
instance MyShow a => MyShow [a] where
  show = ...

-- Implement an ordering for nullable values.
-- `Nothing` should be considered less than any non-null (`Just`)
-- value, and should be equal to itself.
instance MyOrd a => MyOrd (Maybe a) where
  ...
```

Polymorphic Functions

Implement the following polymorphic functions:

```haskell
-- Find the maximum of a non-empty list.
-- Return Nothing is the list is empty.
maximum :: MyOrd a => [a] -> Maybe a

-- Take the sum of a list of elements.
sum :: MyNum a => [a] -> a
```

Good Practice

These are not directly related to the lectures, but will give you a lot of practice with Haskell. These are just a few functions that are used very commonly throughout programming in Haskell, so you should know that they exist and how to implement them.

Thus, implement the following functions:

```haskell
-- Some very convenient list functions!

-- Check if any element satisfies a predicate.
any :: (a -> Bool) -> [a] -> Bool

-- Iterate a function. Yield the infinite list of results.
-- iterate f x = [x, f x, f (f x), ...]
iterate :: (a -> a) -> a  -> [a]

-- Take elements as long as they match a predicate.
takeWhile :: (a -> Bool) -> [a] -> [a]

-- Group elements into sublists by equality.
-- group [1, 1, 2, 2, 3, 3, 2, 3, 5] == [[1, 1], [2, 2], [3, 3], [2], [3], [5]]
group :: Eq a => [a] -> [[a]]

-- A generic version of `group`, which allows the user to specify
-- whether two consecutive elements should be in the same group.
-- group == groupBy (==)
groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
```

Totally Optional: Head Start

The ideas in this section are not just good practice, but will also be in the next set of slides and lecture. However, getting a head start on this will help you understand the material more intuitively, as the next set of slides will be thoroughly abstract and may be somewhat confusing!

I'm including this just in case you have nothing to do, and want to explore a few aspects of Haskell that we'll be getting to soon.

Container Typeclass

You've already seen several things that look like data structures that *hold* another element inside them: namely, you've seen lists and you've seen the `Maybe` type you used above. Sometimes, we want to operate on things within a container without taking them outside the container, and for that we're going to make a Haskell typeclass. This typeclass is going to be a bit different from the previous ones we've looked at. All our previous typeclasses looked like this:

```
class TypeClass a where
    function :: a -> ...
```

Our container typeclass will look like this:

```
class Container c where
    function :: c a -> ...
```

Note that the `function` function does not operate on a value of type `c`, but instead on a value of type `c a`, for some other type `a`. This allows `c` to be something that isn't a full type on it's own, such as `List` or `Maybe`. With that said, here's the typeclass for our containers:

```
class Container c where
  applyInside :: (a -> b) -> (c a -> c b)
```

Your task is to implement this typeclass for two types:

```
-- Same List as you defined on homework 1!
instance Container List where
  applyInside ...

-- Same Maybe as from earlier in this homework.
instance Container Maybe where
  applyInside ...
```

Hint: Take a look at the type signature of `applyInside` when specialized to `List` and compare it to the type signature for `map`! (The phrase "when specialized to `List`" means "when you take `c` and replace it with `List`".)

Make sure the following test code passes (each expression is `True`):

```
-- The function we'll test with.
let square x = x * x

-- Testing on Maybe.
applyInside square Nothing == Nothing
applyInside square (Just 5) == Just 25
```

```haskell
-- Testing on lists.
applyInside square (Cons 3 Nil) == Cons 9 Nil
applyInside square Nil == Nil
applyInside square (Cons 10 (Cons 5 Nil)) == Cons 100 (Cons 25 Nil)
```

Addable Typeclass

Just like we generalized the notion of a container with a container typeclass, we can generalize the notion of types that can somehow be combined, or added, together. However, a type that can *only* be added together is often somewhat uninteresting; it just means we have a function that takes two of some type and outputs another thing of that same type. To make this a bit more interesting, we're going to do a bit more, and require our type to have an *identity element*. The identity element functions like the value zero in normal numeric addition, so we call it `zeroElement`.

This typeclass will look like this:

```haskell
class Addable m where
  zeroElement :: m
  add :: m -> m -> m
```

Note how we have a polymorphic *value* `zeroElement` — this isn't a function, but instead is just a value.

Implement this typeclass for the following data types:

```haskell
data Sum = Sum Int
data FloatSum = FloatSum Int
data List = List [a]
```

Next, add the following instance:

```haskell
instance Addable a => Addable (Maybe a) where
  ...
```

As usual, if a `Nothing` appears anywhere in the addition computation, the result should be `Nothing`; otherwise, it should be `Just` the sum.