Lab 6: Functors, Applicatives, Monads

Functional Programming (ITI0212)

This week we saw some typeclasses that abstract useful functional programming patterns. It will probably take some time to get used to these abstractions, and to notice where they are occuring in your code.

`Functor`s are type constructors `t : Type → Type` supporting a notion of mapping, i.e. implementing a function `map : α → β → t α → t β` which must preserve function composition and the identity function.

`Applicative`s are `Functor`s which additionally supply functions `pure : α → tα` and `seq : t (α → β) → t α → t β`, written infix as `<*>`. `Applicative`s abstract function application to the context of some `Functor`.

`Monad`s are `Applicative`s which supply `bind : t α → (α → t β) → t β`, written infix as `>>=`, which abstracts composition of "effectful" computations (we will see more on this next week).

Task 1
An *homogeneous pair* is an element of type $\alpha \times \alpha$.

We can introduce this as a definition,

```
def HPair (α : Type) : Type := α × α.
```

Define an instance of `Functor` for `HPair`, and convince yourself that it satisfies the two functor laws, i.e.

```
map id p = p
map (f ∘ g) p = map f (map g p).
```

Task 2
Write an instance of `Functor` for $(\alpha \to .)$ `: Type → Type`. You will need to explicitly add α as an (implicit) argument to the instance, i.e. `instance {α : Type} ...`

Hint: work out what the type of `map` is for this `Functor`, and consider how you can use the arguments to produce the output.

Task 3
A `Monoid` is a type equipped with the extra structure given by (copy this definition into your file),

```
class Monoid (α : Type) where
  add : α → α → α
  zero : α
```

These operations must satisfy the equations:

`add zero x = x = add x zero` (*unitality*)

and

`add x (add y z) = add (add x y) z` (*associativity*).

Intuitively, a `Monoid` is a type equipped with some notion of combining any two elements of the type, and with a "null" or "zero" or "empty" element.

Write instances of `Monoid Nat` and `Monoid String` and convince yourself that your implementations of the operations satisfy the equations.

Task 4
Write a function that takes a list of elements from a `Monoid` α, and uses the `Monoid` structure to accumulate the elements in the list into an α.

Task 5
Define a function:

`liftA2 [Applicative t] (f : α → β → γ) (ta : t α) (tb : t β) : t γ`

This function should behave like a generalized version of `map`, e.g. (you will need to copy the definition of 'instance : Applicative List' from the lecture to evaluate this...)

`liftA2 Nat.add [1,2] [3,4]` should evaluate to `[4,5,5,6]` = `[1+3,1+4,2+3,2+4]`.

Hint: use the fact that t is assumed to be an `Applicative`. Recall the types of `pure` and `seq`. How can we use them to compute a `t` γ?

Task 6
Write an instance of `Monad` for `Option`.

Task 7
Copy the following division function that fails if the divisor is zero:

```
def safediv (n m : Nat) : Option Nat :=
if m == 0 then none else some (n / m)
```

Using `safediv` and the `Monad` instance for `Option` defined in the previous task, write a function

`divdiv (p q r: Nat): Option Nat`

which returns (some) $(p/q)/r$ if both q and r are non zero, and none otherwise. Your answer should involve no pattern matching.