Lab 5: Type Classes I

Functional Programming (ITI0212)

This week we saw type classes. You may recognise them as abstract classes in Java. They are like a signature that defines a collection of operations (or methods) that constrain the behaviour of certain generic types.

In this lab, we will practice how to write implementations for type classes as well as how to define our own type classes.

Warm up

Consider a `Shape` type defined as follows

```
inductive Shape : Type where
    | circ : (radius : Float) -> Shape
    | rect : (width height : Float) -> Shape
    | isos : (base height : Float) -> Shape
```

Task 1
Write a `ToString` instance for `Shape`s that outputs the following:

```
> toString (.isos 3 4)
"Triangle with base 3.000000 and height 4.000000"
> toString (.rect 2 3)
"Rectangle with width 2.000000 and height 3.000000"
> toString (.circ 5)
"Circle with radius 5.000000"
```

Notice that Lean's `ToString` instance for `Float`'s rounds the result to 6 decimal places.

Comparing Lists

You have seen in the `Lecture 5` that the default `BEq` implementation for `Lists` compares them *pointwise*,that is, two lists are considered equal if they have the same elements in the same order:

```
> [1,2,3] == [3,2,1]
False
> [1,2,3] == [1,2,3,3]
False
> [1,2,3] == [1,2,3]
True
```

Task 2
Write a named `BEq` implementation for lists that compares them *setwise*: that is, two lists should be considered equal if each element that occurs (at least once) in one of the lists also occurs (at least once) in the other:

```
> [1,2,3] == [3,2,1]
True
> [1,2,3] == [1,2,3,3]
True
> [1,2,3] == [1,2,4]
False
```

Hint : the following functions may be useful:

- `elem : α -> List α -> Bool`
- `all : (α -> Bool)-> List α -> Bool`

Preorders

The `Ord` interface from the standard library allows us to implement *total* orders on the values of a type: an implementation of `Ord` for a given type allows us to compare any two values of that type.

A *preorder* is a more general order relation, which is simply a binary predicate $\leqslant$, having the properties of reflexivity ($\forall x, x \leqslant x$) and transitivity ($\forall xyz, x \leqslant y \wedge y \leqslant z \implies x \leqslant z$).

Note that in a preorder not every two elements are comparable.

Later in the course we will see how to specify these properties in Lean, but for this lab a preorder is just a binary predicate whose implementations we should manually ensure to be reflexive and transitive.

For example, "divides" defines a preorder on the natural numbers: we write $n \leqslant m$ for "n divides m".

Task 3
This task has two parts. First, write a type class `PreOrd` for preorders. Think how many methods you may need.

Then, write a named instance, "divides" for `PreOrd` on `Int` that outputs whether "n divides m". Convince yourself that your implementation is reflexive and transitive.

Hint: you may find the `mod` function useful, where `mod n m` is the remainder when dividing `n` by `m`.

Arithmetic Expressions

Task 4
Recall the type of arithmetic expressions from `Lab 2`:

Write an `Ord` implementation for `AExpr` α that compares the values the arithmetic expressions evaluate to.

Note: for this exercise, you will need to use the `eval` function introduced in the lecture.

Your implementation should behave as follows:

```
> (V 2) < (V 3)
True
> (Plus (V 2) (V 1)) < (V 3)
False
> max (Plus (V 2) (V 4)) (V 3)
Plus (V 2) (V 4)
```

Hint: your implementation will need more than one constraint.

Coe

Task 5
Lean has a type class for coercisons called `Coe` α β. Given a type α, it provides a mechanism for casting it to type β. For example, it is reasonable to cast a `Int` to an `Nat`.

```
instance : Coe Int Nat where
  coe n :=
    match n with
    | Int.ofNat n    => n
    | Int.negSucc _ => 0
```

Write an implementation of the `Coe` type class that coerces a `Bool` to an `Int` and another instance of the same type class that coerces an `Int` to a `Bool`.

Which of the two performs a *lossy* cast? Can you see why?