Lab 5

Functional Programming (ITI0212)

2023-03-03

This week we saw programming interfaces. You may recognise them as typeclasses in Haskell or 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 interfaces as well as how to define our own interface.

Warm up

Recall the `Shape` type from `Lab 2` with the following three constructors:

- `IsoTriangle` (for an isosceles triangle), taking 2 arguments that represent the base and height, respectively.
- `Rectangle` taking 2 arguments representing the width and height, respectively.
- `Circle` which takes 1 argument representing the radius of a circle.

and the following signature:

```
data Shape : Type where
  IsoTriangle : Double -> Double -> Shape
  Rectangle : Double -> Double -> Shape
  Circle : Double -> Shape
```

Task 1
Write a `Show` implementation for `Shape`s that outputs the following:

```
> show (IsoTriangle 3 4)
"Triangle with base 3.0 and height 4.0"
> show (Rectangle 2 3)
"Rectangle with width 2.0 and height 3.0"
> show (Circle 5)
"Circle with radius 5.0"
```

Comparing Lists

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

```
> the (List Nat) [1,2,3] == [3,2,1]
False
> the (List Nat) [1,2,3] == [1,2,3,3]
False
> the (List Nat) [1,2,3] == [1,2,3]
True
```

For the following task you will need to import `Data.List` by writing `import Data.List` at the beginning of your file.

Task 2
Write a named `Eq` implementation for lists that compares them *setwise*:

```
implementation [setwise] Eq a => Eq (List a) where
```

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:

```
> (==) @{setwise} [1,2,3] [3,2,1]
True
> (==) @{setwise} [1,2,3] [1,2,3,3]
True
> (==) @{setwise} [1,2,3] [1,2,4]
False
```

Hint 1: the following functions may be useful:

- `elem : Eq a => a -> List a -> Bool`
- `all : (a -> Bool)-> List a -> Bool`

Hint 2: You may want to use a higher order function that you have seen last week or you may need to use the infix notation for calling Idris functions: (by surrounding it with `backticks`). For example, if you define `add` as in `Lecture 2` you can write `add 3 4` or ``3 `add` 4``.

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 Idris, 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 an interface `PreOrd` for preorders. Think how many methods you may need.

Then, write a named implementation, "divides" for `PreOrd` on `Integer` 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 the lecture:

```
data AExpr : Num n => Type -> Type where
  V : n -> AExpr n
  Plus : AExpr n -> AExpr n -> AExpr n
  Times : AExpr n -> AExpr n -> AExpr n
```

Write an `Ord` implementation for `AExpr n` 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.

Cast

Task 5
Write an implementation of the `Cast` interface that casts a `Bool` to an `Integer` and another implementation of the same interface that casts an `Integer` to a `Bool`.

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