# Lab 11

Functional Programming (ITI0212)

2020.04.29

This week we learned how to interpret equality as an inductively defined indexed type. We saw how case analyzing a premise of equality triggers the discovery that the two things being equated must be the same. We saw how equality interacts with functions via *congruence*, with types via *coercion*, and with indexed types via *transport*. And we introduced the syntactic sugar of *preorder reasoning* for decomposing equality proofs by transitivity into easy-to-understand small steps.

## Multiplication of Natural Numbers

In lecture this week we showed how addition with zero and with a successor behave, and that addition is an associative and commutative operation. In this section you will show some analogous properties of multiplication. You will need to use the facts about addition that we developed in lecture, either by importing the lecture file, or by re-implementing the proofs yourself for practice.

**Task 1**
Convince Idris that 0 is an absorbing element for multiplication on both the left and the right:

```
times_zero_left   :  {n : Nat} -> 0 * n = 0

times_zero_right  :  {n : Nat} -> n * 0 = 0
```

**Task 2**
Convince Idris that multiplication by a successor is the same as repeated addition:

```
times_succ_left   :  {m , n : Nat} -> (S m) * n = (m * n) + n

times_succ_right  :  {m , n : Nat} -> m * (S n) = m + (m * n)
```

**Task 3**
Convince Idris that 1 is a neutral element for multiplication on both the left and the right:

```
times_one_left   :  {n : Nat} -> 1 * n = n

times_one_right  :  {n : Nat} -> n * 1 = n
```

Hint: you should not need to use any case analysis or recursion to do this, simply combine the facts about multiplication from tasks 1 and 2 with some facts about addition from lecture.

**Task 4**
Convince Idris that multiplication is commutative:

```
times_sym  :  {m , n : Nat} -> m * n = n * m
```

## Equalities in Types

Sometimes we need to explain to Idris why a term we write has the type that we claim. This week we learned how to use *coercion* along a type equality and *transport* along an indexed type equality to do just that.

**Task 5**
Complete the definition of the function `twisted_cons` that behaves exactly like the constructor `Data.Vect.(::)`, but has the type shown.

```
twisted_cons  :  a -> Vect n a -> Vect (n + 1) a
```

**Task 6**
Complete the definition of the function that reverses the order of the elements in a vector:

```
reverse_vect  :  Vect n a -> Vect n a
```

Your function should behave as follows:

```
the (Vect _ Integer) (reverse_vect [])  =  []
reverse_vect [1]  =  [1]
reverse_vect [1 , 2 , 3]  =  [3 , 2 , 1]
```