Lab 12

Functional Programming (ITI0212)

2021.04.13

This week we are learning how to interpret equality as an indexed type. In lecture we saw that a term of type `x = y` represents a *proof* that `x` and `y` are equal, while a context variable of type `x = y` represents an *assumption* that `x` and `y` are equal. Because the only constructor for equality types, `Refl`, relates only identical things, case analyzing a premise of equality triggers the discovery that the things being related must be the same.

We saw that all functions preserve equality (*congruence*), that we can write canonical functions between equal types (*coercion*), and in particular between indexed types with equal indices (*transport*). We also introduced *equational reasoning* for decomposing equality proofs by transitivity into easier-to-understand small steps.

Multiplication of Natural Numbers

In this section you will prove some facts about multiplication of natural numbers. You will need to use the facts about addition that we proved 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 the left:

```
times_zero_left   :  (n : Nat) -> 0 * n = 0
```

Task 2
Convince Idris that 0 is an absorbing element for multiplication on the right:

```
times_zero_right  :  (n : Nat) -> n * 0 = 0
```

Hint: if you get stuck, here's a strategy that you can implement by filling the goals:

```
  times_zero_right  :  (n : Nat) -> n * 0 = 0
  times_zero_right Z  =  ?tzr0
  times_zero_right (S n)  =  Calc $
    |~ S n * 0
    ~~ 0 + (n * 0)      ...(?tzr1)
    ~~ 0 + 0            ...(?trz2)
    ~~ 0                ...(?tzr3)
```

Task 3
Convince Idris that multiplying by a successor on the left is the same as repeated addition:

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

Task 4
Convince Idris that multiplying by a successor on the right is the same as repeated addition:

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

Hint: if you get stuck, here's a strategy that you can implement by filling the goals:

```
  times_succ_right  :  {m , n : Nat} -> m * (S n) = m + (m * n)
  times_succ_right {m = Z}  =  ?tsr0
  times_succ_right {m = S m}  =  Calc $
    |~ S m * S n
    ~~ (m * S n) + S n          ...(?tsr1)
    ~~ (m + (m * n)) + S n      ...(?tsr2)
    ~~ S ((m + (m * n)) + n)    ...(?tsr3)
    ~~ S (m + ((m * n) + n))    ...(?tsr4)
    ~~ S m + ((m * n) + n)      ...(?tsr5)
    ~~ S m + (S m * n)          ...(?tsr6)
```

Task 5
Convince Idris that multiplication is commutative:

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

Hint: if you get stuck, here's a strategy that you can implement by filling the goals:

```
  times_sym  :  {m , n : Nat} -> m * n = n * m
  times_sym {m = Z}  =  ?ts0
  times_sym {m = S m}  =  Calc $
    |~ S m * n
    ~~ (m * n) + n        ...(?ts1)
    ~~ (n * m) + n        ...(?ts2)
    ~~ n + (n * m)        ...(?ts3)
    ~~ n * S m            ...(?ts4)
```

Equalities of Types

In this section you will explain to Idris why two types are equal and how to interpret a term as an element of a type equal to its computed type.

Task 6
Use *congruence* to show that the following two types are equal:

```
double_length_vect  :  {n : Nat} -> Vect (2 * n) a = Vect (n + n) a
```

Task 7
Use *transport* to write the function that reverses the order of the elements in a vector:

```
reverse_vect  :  {n : Nat} -> 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]
```