Lab 11

Functional Programming

2024-04-26

This week we are using the *propositions as types* interpretation of logic to represent the *equality relation* as an indexed type family. 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 element constructor for equality types, `Refl`, relates things only to themselves, case analyzing an equality assumption triggers the discovery that the things being related must be the same.

We saw that equality is an *equivalence relation* (*reflexive*, *symmetric*, and *transitive*), and that all functions preserve equality (*congruence*). We also saw that there is a canonical function between equal types (*coercion*), and in particular between indexed types with equal indices (*transport*). Using *equational reasoning* we can decompose equality proofs by transitivity into easier-to-understand small steps.

Natural Number Multiplication

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  :  (m : Nat) -> m * 0 = 0
```

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

```
  times_zero_right  :  (m : Nat) -> m * 0 = 0
  times_zero_right Z  =  ?tzr0
  times_zero_right (S m)  =  Calc $
    |~ S m * 0
    ~~ 0 + (m * 0)      ...(?tzr1)
    ~~ 0 + 0            ...(?tzr2)
    ~~ 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 is 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_comm  :  {m , n : Nat} -> m * n = n * m
```

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

```
times_comm  :  {m , n : Nat} -> m * n = n * m
times_comm {m = Z}  =  ?tc0
times_comm {m = S m}  =  Calc $
  |~ S m * n
  ~~ (m * n) + n      ...(?tc1)
  ~~ (n * m) + n      ...(?tc2)
  ~~ n + (n * m)      ...(?tc3)
  ~~ n * S m          ...(?tc4)
```

Type Equalities

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:

```
> reverse_vect [1 , 2 , 3]
[3, 2, 1]
```