Lab 14

Functional Programming (ITI0212)

2023-05-05

This week we discussed equality types interact with other types. In this session, we will see this in practice, in particular we will get comfortable with using *transport*.

Note: just like last week's lab, do not forget to add `%default total` to the beginning of your script. This will tell Idris that all the functions we are defining should be total, so that we do not accidentally fall into the case in which we can prove any statement by simply giving a function that does not terminate.

Good luck and have fun!

Prerequisites

The terms you will need for this session are listed in the appendix in the last page.

Tasks

Task 1 (`Fin` and sums)
There are as many natural numbers less than $m + n$ as there are natural numbers less than $n + m$.

```
fin_plus_comm : {m, n : Nat} -> Fin (m + n) = Fin (n + m)
```

Task 2 (Appending to a `Vect`)
Provide the following function, which appends an element at the end of a vector.

```
-- vect_append [1, 2, 3] 4 = [1, 2, 3, 4]

vect_append  :  {n : Nat} -> Vect n a -> a -> Vect (S n) a
```

Task 3 (Evenness by adding 2)
In Lecture 11 we presented an inductive predicate characterizing evenness of natural numbers:

```
-- Recall from Lecture 11:

data IsEven : (n : Nat) -> Type where
  IsEvenZ : IsEven 0
  IsEvenSS : IsEven n -> IsEven (S (S n))
```

Prove that if n is even, then $n + 2$ is even as well. You could use proving an intermediate lemma. Try to guess the statement of the lemma as you proceed in the main proof.

```
plus_two_lemma : {n : Nat} -> n + 2 = ?what
```

The main statement that you are asked to prove follows:

```
even_plus_two : {n : Nat} -> IsEven n -> IsEven (n + 2)
```

Task 4 (Cutting out the tail)
You are given a vector of length $(1 + m) \times n$. Since the left factor is at least one, the product is at least n, so it makes sense to ask to cut out the last n elements of the vector.

Again, you will probably need a mysterious lemma. Try and guess how to complete the statement as you work on the main function:

```
mysterious_lemma : {m, n : Nat}
  -> (1 + m) * n = ?some_mysterious_thing
```

You might also want to know that the following function is present in the standard library:

```
-- In the standard library:
-- splitAt : (n : Nat) -> Vect (n + m) elem
--     -> Pair (Vect n elem) (Vect m elem)
```

The main function we ask for has this type signature:

```
cut_tail : {m, n : Nat} -> Vect ((1 + m) * n) a
  -> Pair (Vect (m * n) a) (Vect n a)
```

Task 5 (Optional: matrices)
A `Matrix` is a `Vect` of `Vect`'s. Dependent typing allows us to specify the dimensions of the matrix in its type.

```
Matrix : Nat -> Nat -> Type -> Type
Matrix m n a = Vect m (Vect n a)
```

Here are some functions that we would like you to define.

1. `prepend_row` adjoins a row at the top;
2. `append_row` adjoins a row at the bottom;
3. `prepend_col` adjoins a column on the left;
4. `append_col` adjoins a column on the right.

Hint: to quickly solve subtasks 3 and 4 you might want to use `transpose`, defined in the standard library, whose type signature is also given below.

```
-- transpose : {m, n : Nat} -> Matrix m n a -> Matrix n m a

prepend_row : {m, n : Nat}
  -> Vect n a -> Matrix m n a -> Matrix (S m) n a
append_row  : {m, n : Nat}
  -> Vect n a -> Matrix m n a -> Matrix (S m) n a
prepend_col : {m, n : Nat}
  -> Vect m a -> Matrix m n a -> Matrix m (S n) a
append_col  : {m, n : Nat}
  -> Vect m a -> Matrix m n a -> Matrix m (S n) a
```

Task 6 (Optional: slippery when wet)
This task is for the more intrepid. Below are three functions from the standard library, plus one function I'm giving you now.

```
-- In the standard library:
-- (++) : Vect m elem -> Vect n elem -> Vect (m + n) elem
-- splitAt : (n : Nat) -> Vect (n + m) elem -> Pair (Vect n elem)
    (Vect m elem)
-- concat : Vect m (Vect n elem) -> Vect (m * n) elem
```

```
uncat : {m, n : Nat} -> Vect (m * n) a -> Vect m (Vect n a)
uncat {m = 0} [] = []
uncat {m = S m'} {n = n} v =
  let
    (xs, ys) = splitAt n v
  in xs :: uncat ys
```

Suppose a network protocol API in Idris returns a vector with some amount of packets, which are themselves represented as vectors. The packets are uniform in size, and they consist of a fixed-length header, followed by a fixed-length payload. In types, if h is the header size, and p is the payload size, then the type of packets is `Vect (h + p) a`.

Given a `Vect` of packets, we would like to extract a `Vect` of the headers and a `Vect` of the payloads. Write the function:

```
headers_and_payloads : {n, h, p : Nat} -> Vect n (Vect (h + p) a)
  -> Pair (Vect n (Vect h a)) (Vect n (Vect p a))
```

Appendix

Here are some terms you might need during this session. They were presented during the previous lectures and lab sessions, so they are copied here just for convenience. [1]

```
coerce : a = b -> a -> b
coerce {a = a'} {b = a'} Refl = id

transport : (0 fam : a -> Type) -> {x, y : a}
  -> x = y -> fam x -> fam y
transport fam prf = coerce (cong fam prf)


plus_assoc : {m, n, p : Nat}
  -> (m + n) + p = m + (n + p)
plus_assoc {m = 0} = Refl
plus_assoc {m = (S k)} = cong S plus_assoc

plus_Z : {n : Nat} -> n + 0 = n
plus_Z {n = 0} = Refl
plus_Z {n = (S n')} = cong S plus_Z

plus_S : {m, n : Nat} -> m + S n = S m + n
plus_S {m = 0} = Refl
plus_S {m = S m'} = cong S plus_S

plus_comm : {m, n : Nat} -> m + n = n + m
plus_comm {m = 0} = sym plus_Z
plus_comm {m = (S m')} =
  transitive
    (cong S plus_comm)
    (sym plus_S)

plus_mult_distrib_l : (m, n, p : Nat)
    -> m * (n + p) = m * n + m * p
plus_mult_distrib_l 0 n p = Refl
plus_mult_distrib_l (S m) n p = Calc $
  |~  S m * (n + p)
  ~~  (n + p) + (m * n + m * p) ...( cong (n + p +)
   (plus_mult_distrib_l m n p) )
  ~~  n + (p + (m * n + m * p)) ...( plus_assoc )
  ~~  n + ((p + m * n) + m * p) ...( cong (n +) (sym $ plus_assoc) )
  ~~  n + ((m * n + p) + m * p) ...( cong (\x => n + (x + m * p)) $
   plus_comm)
  ~~  n + (m * n + (p + m * p)) ...( cong (n +) plus_assoc)
  ~~  (n + m * n) + (p + m * p) ...( sym $ plus_assoc)
```

[1] This is like the fifth time I write these.