Lab 3

Functional Programming (ITI0212)

2020.02.26

Recall the type of *vectors* from `Data.Vect` in the Idris standard library:

```
data Vect  :  Nat -> Type -> Type  where
  Nil   :  Vect Z t
  (::)  :  t -> Vect n t -> Vect (S n) t
```

This is an example of a type with both a type *parameter* (the "`t`") and a type *index* (the "`Nat`"). The difference between a parameter and an index is that a parameter must be treated uniformly (both of the constructors result in a `Vect` *something* `t`), whereas an index determines a family of types dependent upon – or *sorted by* – the values of the indexing type (in this case, `Vect`s are sorted by their lengths, which are `Nat`s).

Matrices

Because types are first-class objects in Idris, we can write functions that act on them. As an example of this, we saw in lecture how to define a type of matrices:

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

By this definition, a matrix is a vector of `m` rows, each of which is a vector of `n` cells, each of type `t`. Because the dimensions of a matrix are encoded in its type, we can guarantee that certain well-typed matrix operations will not result in a run-time error. If you have experience working with machine learning libraries in languages with less expressive type systems, you can appreciate how useful this can be.

Because `Vect`s are inductively defined types, we can exploit their structure when working with `Matrix`es; in particular, we can use *pattern matching* and *recursion*.

In order to add two matrices of integers, they must have the same dimensions, and the result of the the addition will also have the same dimensions:

```
add  :  {m , n : Nat} ->
  Matrix m n Integer -> Matrix m n Integer -> Matrix m n Integer
```

Task 1
Complete the definition of matrix addition by the following process:

- Add a skeleton definition to the above type signature, where the entire right-hand side is a hole.
- Case split on both of the two arguments. (How many clauses should there be? How does Idris know that the other possible constructor combinations cannot occur?)

- Use automatic term search to complete the first clause. (Why does this succeed?)
- Define a local variable to hold the result of the recursive call – i.e. adding the tails of the two `Vect`s of `Vect`s.
- Finish the definition by specifying how to add the two head `Vect`s and combining the sum with the result of the recursive call. (Hint: recall the `zipWith` function, discussed in lecture.)

This process of writing a type signature, adding a skeleton definition with a hole, pattern matching on arguments, and recursing on subterms, lies at the heart of type-driven development.

We have defined a `Matrix` as a `Vect` of its rows. However, sometimes we want access to the columns of a matrix instead of the rows. There are several ways we could accomplish this, one of which is to define *matrix transposition*.

Task 2
Use the type-driven development process described above to complete the definition of matrix transposition with the following type signature:

```
transposeMatrix : {t : Type} -> {m , n : Nat} ->
  Matrix m n t -> Matrix n m t
```

- In the clause for the empty vector – i.e. a 0-by-n matrix – the transpose must be an n-by-0 matrix, which is a vector containing n empty vectors. The `replicate` function from the standard library may be helpful here.
- In the clause for a nonempty vector, think about how to transpose all of the rows of a matrix, assuming that you already know how to transpose all of its rows *except for the first*. That assumption corresponds to a recursive function call. The `zipWith` function may again be helpful.

Matrix Multiplication

Recall that to compute the (i, j)-th entry of a matrix multiplication, we take the i-th row of the first matrix and the j-th column of the second matrix (which necessarily have the same length), multiply them pointwise, and sum the result:

$$(A \times B)_{i,j} = \sum_k A_{i,k} B_{k,j}$$

Task 3
Either copy from your lecture notes, or write on your own, the function `sumofproducts` that computes the sum of the pointwise product of two vectors of the same size. Here, you have a choice of either using the inductive structure of `Vect`s directly to write a recursive definition, or using the functions `sum` and `zipWith` in order to write a shorter definition.

```
sumofproducts : {n : Nat} ->
  Vect n Integer -> Vect n Integer -> Integer
```

Now we have all the pieces that we need in order to define matrix multiplication. To perform the matrix multiplication $A \times B$, our strategy will be to use recursion on A *only*.

Task 4
Complete the definition of the matrix multiplication function with the following type signature:

```
mult  :  {m : Nat} ->
  {n : Nat} -> Matrix m n Integer ->
  {p : Nat} -> Matrix n p Integer ->
  Matrix m p Integer
```

Use the following procedure:

- Add a skeleton definition, where the entire right-hand side is a hole.
- Case split on the first argument *only*.
- Complete the first clause using term search. (Why does this succeed?)
- In the second clause, define a local variable `rest_rows` to hold the result of the recursive call (using the tail of the first argument, and the entire second argument unchanged). This will be the matrix containing all of the rows of the product $A \times B$, *except* for the first one.
- We can obtain the j-th entry of the first row of the matrix $A \times B$ by applying `sumofproducts` to the `Vect` that is the first row of matrix A and the `Vect` that is the j-th column of matrix B. Because our matrices are encoded as vectors of rows, we do not have direct access to the columns, but we can obtain them by *matrix transposition*. Use these facts to define a local variable `first_row` whose value is the `Vect` that is the first row of the matrix $A \times B$.
- Write an expression to combine the two local variables `first_row` and `rest_rows` into the matrix consisting of all the rows of the matrix $A \times B$.

Elementary Row Operations

An **elementary row operation** on a matrix is one of the following:

row multiplication: the replacement of a matrix row with the row which is the pointwise product of a nonzero constant and the previous value,

row addition: the replacement of a matrix row with the row which is the pointwise sum of that row and another row of the same matrix,

row swapping: the transposition of two rows of a matrix.

If you are unfamiliar with elementary row operations, you can read more about them here:

`https://en.wikipedia.org/wiki/Elementary_matrix#Elementary_row_operations`

In this lab we will implement the first of these operations, row multiplication. If you have time and interest, please consider working on the other two elementary row operations as well.

The function that computes elementary row multiplication should take as explicit arguments a row index, a factor by which to multiply each element of that row, and a matrix. It should return the matrix in which each element of the given row has been multiplied by the given factor. (For the purposes of this exercise, we do not require the factor to be nonzero.)

In order to guarantee that the row index is valid, we can use the `Fin` type from `Data.Fin` in the standard library that was presented in lecture this week:

```
data Fin  :  Nat -> Type  where
  FZ  :  Fin (S n)
  FS  :  Fin n -> Fin (S n)
```

Notice that the type `Fin n` has exactly `n` elements in it; in particular, the type `Fin Z` is empty. Like `Nat`, the types `Fin n` support the syntactic sugar of Arabic numerals on the *right* hand side of a definition.

There are a few different ways to implement elementary row multiplication in a provably total manner. Here, we will explore one such.

Task 5

- Enter the following type signature for the elementary row multiplication function into Idris and add a skeleton definition with a hole:

  ```
  elem_mult  :  {m , n : Nat} -> Fin m -> Integer ->
    Matrix m n Integer -> Matrix m n Integer
  elem_mult index factor matrix  =  ?Goal
  ```

- Notice that the second argument (called "`factor`" above) is not of an inductive type, so we can't case split on it. (And anyway, it wouldn't make sense to do so because the function doesn't behave differently based on the particular factor that we multiply by.) Try case splitting on the remaining two arguments. What does Idris discover about some of the cases? Why does this make sense? (Hint: think about the implicit argument `m`.)

  Important note: if you case split on the matirix before splitting on the index, you may encounter a bug in Idris where after a case split it can't tell whether "`[]`" refers to the empty vector (`Data.Vect.Nil`) or to the empty list (`Prelude.List.Nil`). The type of the function tells us that the argument `matrix` is a `Vect` of `Vect`s of `Integer`s, so it must be the former. You can work around this bug for now by replacing the "`[]`" in the pattern with "`Data.Vect.Nil`".

- Finish writing the remaining cases to define the `elem_mult` function.