Inf1-FP Revision Tutorial 7

Algebraic types

Week of 30 Nov – 4 Dec 2015

All of the following questions are taken directly from past exam papers. To check your solutions write tests that they behave according to the examples given, plus any QuickCheck tests you can think of. You will need to start from the template file, which contains the code required to drive QuickCheck on the data types used in these questions.

1. The following data type represents expressions built from variables, sums, and products.

```
data Expr = Var String
          | Expr :*: Expr
          | Expr :+: Expr
```

   (a) Write two functions `isTerm, isNorm :: Expr -> Bool` that return true when the given expression is a term or is normal, respectively. We say that an expression is a *term* if it is a product of variables, that is, it is a variable or the product of two expressions that are terms. We say that an expression is *normal* if it is a sum of terms, that is, if it is a term or the sum of two expressions that are normal. For example,

```
isTerm (Var "x")                                          =  True
isTerm ((Var "x" :*: Var "y") :*: Var "z")                =  True
isTerm ((Var "x" :*: Var "y") :+: Var "z")                =  False
isTerm (Var "x" :*: (Var "y" :+: Var "z"))                =  False

isNorm (Var "x")                                          =  True
isNorm (Var "x" :*: Var "y" :*: Var "z")                  =  True
isNorm ((Var "x" :*: Var "y") :+: Var "z")                =  True
isNorm (Var "x" :*: (Var "y" :+: Var "z"))                =  False
isNorm ((Var "x" :*: Var "y") :+: (Var "x" :*: Var "z"))  =  True
isNorm ((Var "u" :+: Var "v") :*: (Var "x" :+: Var "y"))  =  False
isNorm (((Var "u" :*: Var "x") :+: (Var "u" :*: Var "y")) :+:
        ((Var "v" :*: Var "x") :+: (Var "v" :*: Var "y")))  =  True
```

   (b) Write a function `norm :: Expr -> Expr` that converts an expression to an equivalent expression in normal form. An expression not in normal form may be converted to normal form by repeated application of the distributive laws,

$$\begin{aligned}(a+b)\times c &= (a\times c)+(b\times c)\\ a\times(b+c) &= (a\times b)+(a\times c)\end{aligned}$$

For example,

```
norm (Var "x")
  =  (Var "x")
norm ((Var "x" :*: Var "y") :*: Var "z")
  =  ((Var "x" :*: Var "y") :*: Var "z")
norm ((Var "x" :*: Var "y") :+: Var "z")
  =  ((Var "x" :*: Var "y") :+: Var "z")
norm (Var "x" :*: (Var "y" :+: Var "z"))
  =  ((Var "x" :*: Var "y") :+: (Var "x" :*: Var "z"))
norm ((Var "u" :+: Var "v") :*: (Var "x" :+: Var "y"))
  =  (((Var "u" :*: Var "x") :+: (Var "u" :*: Var "y")) :+:
      ((Var "v" :*: Var "x") :+: (Var "v" :*: Var "y")))
```

2. (a) A scalar is a single integer, and a vector is a pair of integers.

```
type Scalar = Int
type Vector = (Int,Int)
```

Write functions

```
add :: Vector -> Vector -> Vector
mul :: Scalar -> Vector -> Vector
```

that add two vectors by adding corresponding components of the vectors, and multiply a scalar and a vector by multiplying each component of the vector by the scalar. For example,

```
add (1,2) (3,4) == (4,6)
mul 2 (3,4)     == (6,8)
```

(b) The following data type represents terms that compute vectors. A term is a vector consisting of two scalars, the sum of two terms, or the multiplication of a scalar by a term.

```
data Term  =  Vec Scalar Scalar
           |  Add Term Term
           |  Mul Scalar Term
```

Write a function `eva :: Term -> Vector` that takes a term and computes the corresponding vector. For example,

```
eva (Vec 1 2)                                  ==  (1,2)
eva (Add (Vec 1 2) (Vec 3 4))                  ==  (4,6)
eva (Mul 2 (Vec 3 4))                          ==  (6,8)
eva (Mul 2 (Add (Vec 1 2) (Vec 3 4)))          ==  (8,12)
eva (Add (Mul 2 (Vec 1 2)) (Mul 2 (Vec 3 4)))  ==  (8,12)
```

(c) Write a function `sho :: Term -> String` that converts a term to a string. Vectors should be printed as a pair of integers in parentheses, sums and products should be written infix surrounded by parentheses. For example,

```
sho (Vec 1 2)                          ==  "(1,2)"
sho (Add (Vec 1 2) (Vec 3 4))          ==  "((1,2)+(3,4))"
sho (Mul 2 (Vec 3 4))                  ==  "(2*(3,4))"
sho (Mul 2 (Add (Vec 1 2) (Vec 3 4)))  ==  "(2*((1,2)+(3,4)))"
sho (Add (Mul 2 (Vec 1 2)) (Mul 2 (Vec 3 4)))
                                       ==  "((2*(1,2))+(2*(3,4)))"
```

You may use the `show` function on scalars in your definition, but *not* the `show` function on terms that is provided in the template file for use by QuickCheck.

3. We introduce a data type to represent collections of points in a grid:

```
type Point = (Int,Int)
data Points = Rectangle Point Point
            | Union Points Points
            | Difference Points Points
```

The grid starts with `(0,0)` in the top left corner. The first coordinate of a point represents the horizontal distance from the origin, the second represents the vertical distance.

The constructor `Rectangle` selects all points in a rectangular area. For example,

```
Rectangle (0,0) (2,1)
```

gives the top left and bottom right corners of a rectangle, and represents all the points in between (inclusive):

```
(0,0)  (1,0)  (2,0)
(0,1)  (1,1)  (2,1)
```

Secondly, `Union` combines two collections of points; for example,

```
Union (Rectangle (0,0) (1,1)) (Rectangle (1,0) (2,1))
```

represents the same collection of points as above. Finally, the constructor `Difference` selects those points that are in the first collection but not in the second. For example:

```
Difference (Rectangle (0,0) (2,2)) (Rectangle (0,2) (3,2))
```

again gives the same collection of points as above.

(a) Write a function

```
inPoints :: Point -> Points -> Bool
```

to determine whether a point is in a given collection. For example:

```
inPoints (1,1) (Rectangle (0,0) (2,1))  == True
inPoints (3,4) (Rectangle (0,0) (2,1))  == False
inPoints (1,1) (Union (Rectangle (0,0) (0,1))
                      (Rectangle (1,0) (1,1))) == True
inPoints (2,2) (Union (Rectangle (0,0) (0,1))
                      (Rectangle (1,0) (1,1))) == False
inPoints (1,1) (Difference (Rectangle (0,0) (1,1))
                           (Rectangle (0,0) (0,1))) == True
inPoints (0,0) (Difference (Rectangle (0,0) (1,1))
                           (Rectangle (0,0) (0,1))) == False
```

(b) Write a function

```
showPoints :: Point -> Points -> [String]
```

to show a collection of points as a list of strings, representing the points on a grid. The grid starts with `(0,0)` in the top left corner, while the bottom right corner, which determines the size of the grid, is given by the first argument to the function `showPoints`. The strings in the list that is returned should correspond to the rows (not the columns) of the grid. Use an asterisk (’*’) to represent a point, and use blank space (’ ’) to fill out the lines. For example:

```
showPoints (4,2) (Rectangle (1,1) (3,3)) ==
  ["     ",
   " *** ",
   " *** "]
showPoints (5,2) (Difference (Rectangle (0,0) (4,1))
                             (Rectangle (2,0) (2,2))) ==
  ["** ** ",
   "** ** ",
   "      "]
```