Inf1-FP Revision Tutorial 8

General revision

Week of 7 – 11 Dec 2015

In this tutorial we will quickly go through all of the major topics of the course. Since this is very general revision, you should not have any trouble doing any of these exercises. If you do find any of them challenging, perhaps you should focus on revising that area.

1. Functions

Define a function `f1` that given two numbers will tell us whether the first is divisible by the second. You should provide an appropriate type declaration for this function.

2. Recursion

Define a recursive function `f2` that given two lists of numbers will give the product of all the numbers in the first list that are divisible by the corresponding number in the second list.

```
f2 [21, 34, 22, 9] [2, 2, 4, 3] == 34 * 9 == 306
```

Your function should give a meaningful error if the lengths of the lists do not match.

You should also have an idea how you would approach this problem using both list comprehension and higher order functions.

3. Characters and Strings

Write a function `f3` that given a string containing only letters and spaces, will return a list of the words in the string that begin with an uppercase letter. For example:

```
f3 "this is   a String of  Words" == ["String","Words"]
```

A library function might come in handy, but you don't need to use it.

4. List Comprehensions

Given two lists of integers, use list comprehension to define a function `f4` that returns a list of all the possible sums of a number from the first list and a number from the second list:

```
f4 [1, 2] [3, 4] == [4,5,5,6]
f4 [1,2] [5,6,9,2] == [6,7,10,3,7,8,11,4]
```

5. List Processing with Higher Order Functions

Using `foldr` give definitions of `map'` and `filter'` that behave like their prelude counterparts.

Also think how you would solve 2, 3 and 4 above with higher order functions.

6. Algebraic Data Types

Write a definition for the type `TwoTree a` where a branch can have 0, 1 or 2 children, together with a node label of type `a`.

Write a function `treeFold :: (a -> b -> b) -> b -> TwoTree a -> b`, that will work similarly like `foldr` but for your `TwoTree` type. That is, the function should go through all the nodes of the tree and apply the function passed to `treeFold` on all of these (the order in which you go through them isn't specified).

7. Type classes

Create a type class called `StructurallySimilar` which defines a function `similar :: a -> a -> Bool` that checks if the two arguments have the same structure.

Make your `TwoTree` an instance of this class. Two `TwoTree`s are `similar` iff each node has the same number of children as the corresponding node in the other tree.

Hint: Lists would be similar iff they had the same number of elements. We can do this by:

```
instance StructurallySimilar [a] where
  similar l1 l2 = length l1 == length l2
```

Then also make `TwoTree` an instance of the `Show` class.

8. Testing

Import `Test.QuickCheck` into your file. Write tests that verify if the following versions of the well known library functions work correctly (if possible, try to think of tests other then just checking if they behave identically to their prelude counterparts). Fix any bugs that you find:

```
product' []     = 0
product' (x:xs) = x * product' xs

x < y = not (x > y)
```

9. Modules

Package the code you have written in 6 and 7 into a module, and check that it still works.

by Jakub Hampl, 7 Dec 2012