ST: Lab Assignment 2

Elias El Khaldi Ahanach, Dylan Bartels, Wojciech Czabanski, Quinten Heijn

September 17, 2017

1 Random floating point numbers

The test should only look at the numbers given by the quartiles and can therefore not completely assure that the method of Curry is actually random. An example of this would be the set of [0.1, ... , 0.1 , 0.3, ... , 0.3, 0.6 , .. 0.6 , 0.8 , .. , 0.8]. This set is of size 1000 and results in a perfect distribution of 250 per quartile. However, it is not random at all.
Keeping this in consideration, we still implemented a test that can somewhat tell if the the set is random based solely on the distribution array. The optimal array should be [n/4,n/4,n/4,n/4] (n/4 per quartile).
Since small values of n can give non-trustworthy distributions, for the test we will only focus on sets of the size >= 1000. The way the test determines whether the distribution can relate is by checking if the distribution values differ no more than 5 percent (a = 0.05) from their optimal values. For instance, for n = 1000, the values of the distribution must be between 237.5-262.5.
The implementation of this test can be found in the runTest function.

2 Recognizing triangles

The *triangle* function returns the type of the matched triangles based on the length of the sides. Since the domain of all the arguments is the set of integral numbers we can order the properties, so that we can be sure that the triangle gets qualified to the most specific group possible (for example is a triangle is **Equilateral** it is also **Isosceles**, but the function will qualify it as **Equilateral**). Other groups are mutually exclusive so they do not pose a problem (for example **Rectangular** and **Equilateral**)

3 Testing properties strength

4 Recognizing Permutations

There are two properties we want to check for *isPermutation*.

1. All permutations of a list should evaluate **True**.
2. All other list should evaluate **False**.

The first property is easily tested by generating all the properties of a list. This can be done with the Haskell function *permutations*. The second property we can test by generating all possible combinations of a list and removing the possible permutations. The combination can be generated be taking all the permutations of all the *subsequences*. However, because the amount of combinations grows really quickly, you can only test for small input lists. This is why we created a third test function that just takes a list and a second list of random list that shouldn't be permutations of the first list. This is ensured by removing all permutations of the first list from the second list. This way, testing can also be automated.

5 Recognizing and generating derangements

These are the properties we could use to test *isDerangement* (from weak to strong):

1. It should only return **True** when the lists are equal in size.
2. It should only return **True** when the lists are permutations of each other.
3. It should only return **True** when the second list is a derangement of the first list and only **False** when this is not the case.

We can automate the testing process because we can easily create lists that are derangements and list that are not derangements based on an input list. To do this we should use random number generation, making the implementation somewhat cumbersome (because of the mixing of pure and unpure functions).

6 Implementing and testing ROT13 encoding

ROT13 is a simple Ceasar cipher that shifts every letter in the input by 13. The special thing about this cipher is that because of the 13-step shift, it is symmetric (rot13(rot13(a)) == a).
The implementation can be found in the .hs file.
The obvious way of testing the ROT13 function would be checking whether all letters have actually shifted 13 steps, but since that would be similar to implementing the method again, we formulated some properties that should hold for the function:

1. Length of input should equal the length of the output.
2. The input (if non-empty) should be different that the output (ROT13(a) != a).
3. The function is symmetric, meaning it returns itself after applying ROT13 twice (ROT13(ROT13(a)) == a).
4. The interval between each letter of the input should be the same for the output (e.g. for "aab" -(ROT13)-> "nno" the interval is [0,1]).
5. The frequencies should remain the same. Even though the letters change, the frequency array should stay the same (e.g. "aab" -(ROT13)-> "nno", the frequency array for both the input and output is [2,1]).

Note that the length property did not have to be explicitly implented since the interval property can only hold if the lengths of the input and output are the same. Therefore checking the interval property also checks the length property.

7 Implementing and testing IBAN validation

There are two properties used to check if string is a valid IBAN number:

1. The length of the IBAN number is the correct length for its country listed in the appropriate ISO document.[1]
2. The sum of the digits of the IBAN number, with letters being converted to numbers according to the following scheme: A -> 10, B -> 11, ..., Z -> 35, modulo 97 is equal to 1.

The first property checks if the number is not missing any digits or contains superfluous digits. The second property validates that the single digits have not been modified. Both properties need to yield **True** for the string to be a valid IBAN number.

7.1 Testing

Testing of the properties can be achieved as follows:

1. Preparing a test list of IBAN numbers with correct country codes and adding a few incorrect country codes and numbers with correct country codes but incorrect lengths.
2. Preparing a test list of 97 identical IBAN numbers with a different check digit (ranging from 02 to 98). The checksum property should discard 96 incorrect IBAN numbers and yield **True** for the IBAN number containing the correct check number.

[1] IBAN Registry

Automating the testing process Automating the test process is difficult because the number of valid alphanumeric combinations is relatively small, so it cannot be entirely randomized and the check digits cannot be calculated by the test generator, because that would mean using the same logic that is being tested.