Lab 3 - Week 5. Mutation Testing

Aim

In this lab, we write our own mutation testing framework like FitSpec. We manipulate the output data of a function under test, to check that any incorrect output for a given input fails the tests. This way, we can discover survivors, the minimal set of properties, and find all the conjectures.

For this lab, we use a 'multiplication table' implementation+properties as target for mutation testing. The related Haskell code can be found in *MultiplicationTable.hs*.

Prerequisites

- Take a look at the paper FitSpec: refining property sets for functional testing.
- Look at the example code in the `Mutation.hs` file.
- Take a look back at the weaker/stronger theory of the Hoare lab.
- Look at the example code in the `FitSpec.hs` file and run the *main* function.

Submission Guidelines

- Submit one Haskell file per exercise (only Haskell files allowed).
- Name each file as `ExerciseX.hs` for exercises and `EulerX.hs` for Euler problems.
- Ensure that the respective module of each file has the same naming format (`module ExerciseX where`).
- Follow the exercise naming conventions closely. Some exercises go through automated testing, so it is important not to change the indicated declarations.
- Do not include any personally identifiable information in the submissions.
- If using additional dependencies, indicate so in a comment at the top of the file.
- Indicate the time spent on each exercise (**Time Spent: X min**).
- Codegrade: For each assignment, you need to create a new group with all teammates. Please note that once you submit, you cannot change the team structure, so be cautious.

Imports

```
import Data.List
import Test.QuickCheck
import Mutation
```

Exercise 1

We provide some mutators to mutate the output of the list in Mutation.hs. Write down which types of output are not yet covered by these mutators, and about their weakness/strength. Come up with a list of other mutators and implement (a subset of) them.

Hint: A mutant can never be the same as the original output list.

Deliverables: List of mutators and rationale, implementation, indication of time spent.

Exercise 2

Write a function countSurvivors that counts the number of survivors:

```
countSurvivors :: Integer -> [([Integer] -> Integer -> Property)] -> (Integer -> [Inte
ger]) -> Integer
```

Where the first argument is the number of mutants (4000 in the FitSpec example), the second argument is the list of properties, and the third argument is the function under test (the multiplication table function in this case).

The output is the number of surviving mutants (0 in the FitSpec example).

Document the effect of which mutations are used and which properties are used on the number of survivors.

Hints:

1. Consider the relation between properties and survivors.
2. The above-mentioned function definition is not final. Feel free to modify it, for example by adding the mutations that should be used.

Deliverables: implementation, documentation of approach, effect of using different mutators/properties, indication of time spent.

Exercise 3

Implement a function that calculates the minimal property subsets, given a 'function under test' and a set of properties.

Deliverables: implementation, documentation of approach, indication of time spent.

Exercise 4

Implement a function that calculates the strength of a given set of properties, which is the percentage of mutants they kill.

Deliverables: implementation, documentation of approach, indication of time spent.

Exercise 5

Implement function(s) that calculate the conjectures: properties that are equivalent, whose cases are subsets of other properties, etc.

Deliverables: implementations, documentation of approach, indication of time spent.

Exercise 6 (Bonus)

Create a function that we pass the function under test, a set of properties, and a number indicating the number of mutants, that visualizes the results of the functions from the previous exercises.

Deliverables: Implementation, indication of time spent.