Advanced Programming (I00032)
Testing stateless systems with logical properties

Assignment 6

This assignment consists of two parts:

1. you extend the `unitTest` module that has been explained in the lecture, and which implementation has been included on Blackboard;
2. you use the extended `unitTest` module to design tests for a given implementation of rational numbers, the source of which is also included on Blackboard.

Important notes: do not use the special *iTask* environment that has been used for assignments 4 and 5 because these use and require slightly different generic implementations. Use the Clean compiler and environment of the first three assignments. Use the environment *Gast* for the correct paths and compiler settings.

1 Extension of `unitTest`

1.1 Custom tests

Extend the `unitTest` module with two new test functions:

```
testPred1 :: String (x → Bool)  x   → TestFun | genShow {|*|}  x
testPredL :: String (x → Bool) [x] → TestFun | genShow {|*|}  x
```

The function `testPred1` tests the given predicate for the given argument. If the test fails the system reports the problem including the argument `x` that raised the issue. The function `testPredL` tests the given predicate for each value in the given list of arguments. Please note that with the available primitives in `unitTest`, the implementation of these new test functions can be concise (one or two lines of code for each new function).

1.2 Generic tests

The new test functions of part 1.1 expect the tester to come up with test values. It is more interesting to implement a `testPred` test function that generates test arguments itself if they are needed. Just like Gast we interpret the boolean `True` as a successful test and `False` as an issue that has to be reported. Here are some examples of predicates that we wish to test automatically in this way:

```
pNot       :: Bool → Bool
pNot b     = not (not b) == b

pReverse   :: [Bool] → Bool
pReverse l = reverse (reverse l) == l

pPlus      :: Int Int → Bool
pPlus x y  = y + x == x + y
```

Functions are interpreted as universally quantified properties over *finite* values of the given domains in the type signature. For instance, the function `pNot` states that for all defined Booleans `b` the negation of the negation of `b` is equal to `b`. Similarly, `pReverse` states that for all *finite* lists, applying the reverse function twice is the identity for these lists.

Just like Gast we make a class to test these properties:

```
class testPred a :: [String] a → TestFun
```

We use here a list of strings rather than a single string to collect information of the supplied arguments. The `Bool` instance of this class gives the base case. The more interesting instance of this class is for functions. For a function you have to generate arguments using the generic functions from Gast. Typically an expression generating a test suite (list of test data) looks like `ggen {|*|} 2 aStream`. For the first N arguments you test the property you get by applying the predicate to such an argument. In contrast with Gast it is not necessary to do fancy things with diagonalisation of test cases. This should enable you to execute programs like:

```
Start
 = doTest
    ( test1                              // custom tests for rot13
    ‘ test2                              // custom tests for rot13
    ‘ test3                              // custom tests for pow
    ‘ testPred ["pNot:"]     pNot        // generic tests for pNot
    ‘ testPred ["pReverse:"] pReverse    // generic tests for pReverse
    ‘ testPred ["pPlus:"]    pPlus       // generic tests for pPlus
    )
```

2 Properties for Rational numbers

On Blackboard you find an implementation of rational numbers, `Rat`. A rational number consists of two integers: the nominator `n` and the denominator `d`.

```
:: Rat = { n :: !Int, d :: !Int }
```

These numbers are always normalized using a greatest common divisor algorithm. Hence, instead of storing 2 / 4 the value 1 / 2 is stored. The denominator `d` should always be a positive number. The internal function `simplify` of this module takes care of this normalization.

Unfortunately, the implementation of the `Rat` module contains errors. Use the extended `unitTest` module to design and implement tests for the operators + and − for `Rat`. Give properties of rational numbers such that each operator is used at least in one property. Write enough properties to spot at least 3 errors in the given implementation of the operators. Note that this can require less or more than three properties that produce a counterexample. For instance two different issues (failing tests) can be caused by the same error (bug in the program).

Errors only have to be corrected if they prevent the execution of other tests. The focus of this assignment is on writing properties and testing, not on providing a correct implementation of rational numbers.

Deadline

The deadline for this exercise is October 21, 23:59h. Upload your extended `unitTest` module and the test suite that you have developed to detect the issues in the `Rat` module.