185.A05 Advanced Functional Programming SS 2020

Monday, 11 May 2020

Assignment 5

on Chapter 5, Chapter 6, Chapter 12, Chapter 8

Topics: Testing, Verification, Monadic Programming, Abstract Data Types

Submission deadline: Monday, 18 May 2020, 12am

Regarding the deadline for the second submission: Please, refer to „Hinweise zu Organisation und Ablauf der Übung“ available at the homepage of the course.

Important:

1. Carefully read and follow the instructions outlined in the complementary files provided with assignment 1. If you have any questions regarding these instructions, ask your questions in the TISS forum. Following these instructions is paramount to ensure a smooth processing of your submitted file with the test system.
2. Store all functions to be written for this assignment in a top-level file named

   `Assignment5.hs`

   of your group directory starting with the module declaration

   `module Assignment5 where`

   Comment your program meaningfully; use auxiliary functions and constants, where reasonable. The very same file name shall be used for the second submission of assignment 5.
3. *Do not use self-defined modules!* If you want to re-use functions (written for other assignments), copy them to `Assignment5.hs`. Import declarations for self-defined modules will fail: Only `Assignment5.hs` will be copied for the (semi-automatical) evaluation procedure, no other ones.
4. Your programs will be (semi-automatically) evaluated on `g0` using the there installed GHC interpreter of GHC version 8.65. If you use a different tool or a different version of GHC for program development, please, double-check well in time before the submission deadline that your programs behave on `g0` and the there installed GHC interpreter version as you expect.

Programming tasks:

1. The Fibonacci numbers F_i, $i \geq 0$, are given by the recursion:

   $$F_0 = 0,\ F_1 = 1, F_{n+2} = F_{n+1} + F_n \quad \text{for } n \geq 0$$

   Consider claim C:

   $$\forall\, n > 1.\ F_{n+1}F_{n-1} - F_n^2 = (-1)^n \qquad (C)$$

1.1 **Without submission:** Check manually for some values, if claim C could hold.

1.2 Implement QuickCheck properties allowing to challenge the validity of claim C automatically and more thoroughly than manually using an efficient implementation of the Fibonacci function of your choice:

```
fib :: Integer -> Integer
fib ...
```

Using your implementation of `fib`, define properties `prop_ccc_...` for testing, if claim C can be valid, where '`ccc`' shall remind to 'checking claim c.' In detail, the property with postfix

- `1` shall do this in the most straightforward way supported by QuickCheck, i.e., every integer generated by QuickCheck shall be used as a test input.
- `2` shall consider only positive integers greater than 1 as test inputs using a precondition for filtering test case candidates generated by QuickCheck accordingly.
- `3` shall use a generator to ensure that only positive integers greater than 1 are generated as test inputs by QuickCheck.
- `4` shall ensure that only test data n within the range of $1000 \leq n \leq 2000$ are used as test inputs by filtering generated test data accordingly by means of an appropriate precondition.
- `5` shall ensure that only test data n within the range of $1000 \leq n \leq 2000$ are generated by QuickCheck.

```
prop_ccc_1 :: Integer -> Bool
prop_ccc_1 n = ...
prop_ccc_2 :: Integer -> Property
prop_ccc_2 n = ...
prop_ccc_3 :: Integer -> Property
prop_ccc_3 n = ...
prop_ccc_4 :: Integer -> Property
prop_ccc_4 n = ...
prop_ccc_5 :: Integer -> Property
prop_ccc_5 n = ...
```

1.3 Use the reporting features of QuickCheck to get more detailed information about the test data generated and used. To this end extend the implementation of `prop_ccc_3` using the QuickCheck combinators `trivial`, `classify`, and `collect`, respectively. Using the combinator

- `trivial`, `prop_ccc_3_trivial` shall report the percentage of *trivial* test inputs. As *trivial* we consider test inputs smaller or equal to 10. A possible report could thus be:

  `OK, passed 100 tests (26% trivial).`

- `classify`, `prop_ccc_3_classify` shall report the percentages of test inputs in the ranges $2 \leq$ test input ≤ 10, $11 \leq$ test input ≤ 100, and $101 \leq$ test input. A possible report could thus be:

```
OK, passed 100 tests.
29% of test inputs in the range [2..10].
33% of test inputs in the range [11..100].
38% of test inputs in the range [101..].
```

- `collect`, `prop_ccc_3_collect` shall report the percentages of all test inputs, i.e., the histogram of test inputs. An excerpt of a possible report could thus be:

```
OK, passed 100 tests.
4% 2.
1% 3.
2% 4.
6% 5.
...
1% 34.
...
```

1.4 **Without submission:** Did you gain sufficient confidence that claim C could actually be valid thanks to manual and automatic testing? If so, start proving claim C formally. What proof principle seems (most) appropriate for this purpose? Why?

1.5 Consider claim D, where t denotes some threshold value:

$$\begin{array}{ll} F_{n+1}F_{n-1} - F_n^2 = (-1)^n & \text{if } 2 \leq n \leq t \\ F_{n-1}F_n - F_{n+1}^2 = (-1)^n & \text{if } t+1 \leq n \end{array} \qquad (D)$$

Modify `prop_ccc_2` for checking claim D. The new property shall be named `prop_ccd` and ensure by test case filtering that only threshold values and test data greater than 1 are used as test inputs.

```
type Threshold = Integer
prop_ccd :: Threshold -> Integer -> Property
prop_ccd t n = ...
```

Experiment with the new property `prop_ccd`.

2. Intuitively, greedy search is supposed to yield short unit fractions of a fractional number, i.e., unit fractions consisting of a small number of summands. As we have seen in assignment 4, this is not always true. In order to get a better feeling for the length of unit fractions generated by greedy search, we want to use QuickCheck.

2.1 Define a property `prop_greedy` which shall be falsified if the number of summands of the unit fraction of a fractional number yielded by greedy search properly exceeds some maximum number. Use generators to ensure that the generated `MaxNumberOfSummands` values are in the range of $2 \leq max \leq 5$ and the generated `Numerator` and `Denominator` values are greater

than 1. Filter additionally the generated pairs of (Numerator,Denominator) values for those where the numerator value is properly smaller than the denominator value.

```
type MaxNumberOfSummands = Integer
type Numerator           = Integer
type Denominator         = Integer
prop_greedy :: MaxNumberOfSummands -> (Numerator,Denominator) -> Property
prop_greedy max (n,d) = ...
```

Experiment with pop_greedy.

Note: Do not use a module import for reusing your greedy search implementation of assignment 4; instead, copy the required parts to the submission file Assignment5.hs.

2.2 **Without submission:** Does the check more often fail questioning the intuition about greedy search based unit fraction decomposition of fractional numbers or more often succeed supporting it? If the property can be falsified, how many test data does it take on average to find a counterexample? Experiment also with other value ranges for MaxNumberOfSummands values.

3. Integer stacks can (naively) be implemented in terms of lists:

```
type Stack  = [Integer]
empty       = []
is_empty [] = True
is_empty _  = False
push x xs   = (x:xs)
pop []      = error "Stack is empty"
pop (_:xs)  = xs
top []      = error "Stack is empty"
top (x:_)   = x
```

The above implementation is a correct implementation of integer stacks iff the operations satisfy the laws 3.1,...,3.6:

```
3.1  is_empty empty      == True
3.2  is_empty (push v s) == False
3.3  top empty           == undefined
3.4  top (push v s)      == v
3.5  pop empty           == undefined
3.6  pop (push v s)      == s
```

Obviously, the implementations of top and pop satisfy law 3.3 and 3.5, respectively. Implement properties prop_31, prop_32, prop_34, and prop_36 allowing to test that the operations in charge obey the laws 3.1, 3.2, 3.4, and 3.6, too:

```
prop_31 :: Bool
prop_31 = ...
```

```
prop_32 :: Integer -> Stack -> Property
prop_32 n ns = ...
prop_34 :: Integer -> Stack -> Property
prop_34 n ns = ...
prop_36 :: Integer -> Stack -> Property
prop_36 n ns = ...
```

Self-defined generators for integer or stack values are not required but differently detailed reports. Property

- `prop_31` shall just deliver the default report.
- `prop_32` shall indicate the percentage of trivial test inputs. A test input is considered trivial, if it involves the empty stack or a stack with a single entry. A report could thus be:

  ```
  OK, passed 100 tests (24% trivial).
  ```

- `prop_34` shall indicate the percentages of test inputs involving the empty stack, one-entry stacks, two-entry stacks, and stacks with more than two entries. A report could thus be:

  ```
  OK, passed 100 tests.
  37% of test inputs: the empty stack.
  28% of test inputs: a one-entry stack.
  12% of test inputs: a two-entry stack.
  23% of test inputs: a non-trivial stack.
  ```

- `prop_36` shall yield a histogram of the number of entries of the stacks involved in the test inputs. A report could thus be:

  ```
  OK, passed 100 tests.
  34% 0.
  25% 1.
  18% 2.
  12% 4.
  11% 6.
  ```

Without submission: Why should the implementation of stacks in this excercise be considered ‘naive?’ Why should it be considered inadequate and inappropriate for usage in ‘real-world’ Haskell programs?

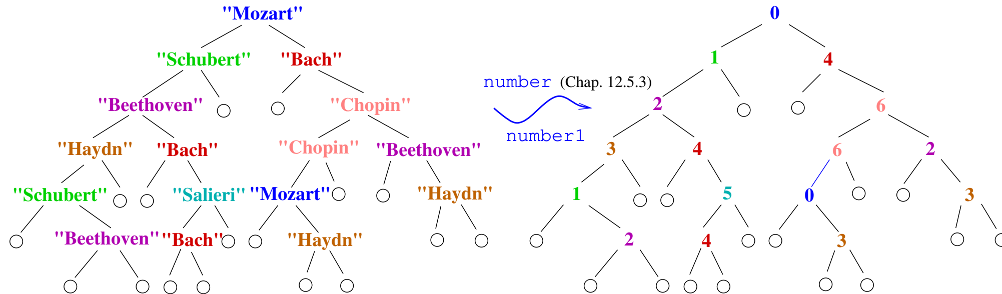
4. *Testing monadic and non-monadic programs:* We consider the problem of node label renaming of Chapter 12.5.3:

   ```
   data Tree a = Nil | Node a (Tree a) (Tree a) deriving Show
   ```

   4.1 *Monadic programming:* Implement the renaming function

   ```
   number :: Eq a => Tree a -> Tree Int
   ```

as shown in Chapter 12.5.3 using the state monad. Make sure that you understand how it works. Note that `number` replaces a node label by the smallest free number, i.e., not yet used number when the label is first reached in the course of a prefix traversal of the tree. For illustration consider the below figure:



4.2 *Non-monadic programming:* Implement a renaming function

```
number1 :: Eq a => Tree a -> Tree Int
```

which is functionally equivalent to `number` but does not use monadic programming.

4.3 Define a property

```
prop_rename :: Tree String -> Property
prop_rename ...
```

allowing to test, if `number` and `number1` are indeed functionally equivalent when applied to trees labelled with strings. Make sure that trees generated as test data do not become too large. To this end use weights 1 and 3 for generating the trivial tree `Nil` or a nontrivial tree starting with constructor `Node`, respectively, when making `Tree a` an instance of type class `Arbitrary`. The report generated by `prop_rename` shall include the information on the percentage of trivial test cases, where a test case is considered trivial iff it is equal to the trivial tree `Nil`.

4.4 **Without submission:** Experiment with different values for the weights used in the `Arbitrary` instance declaration of `Tree a`. Can you get more detailed information on the size of trees generated and used as test cases?

Iucundi acti labores.
Getane Arbeiten sind angenehm.

Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller