МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности
Высшая школа технологий искусственного интеллекта
Направление 02.03.01 Математика и компьютерные науки

Отчет по лабораторной работе №4

по дисциплине «Функциональное программирование»

Вариант 4

Обучающийся: _____ Шихалев А.О.

Руководитель: _____ Моторин Д.Е.

«_____»_____20____г.

Санкт-Петербург, 2024

Содержание

Введение

Практическое задание №4 представляет собой реализацию следующего заданий:

1. **Функция 1:** Написать функцию сложения по модулю addMod :: Int -> Int -> Int -> Int, которая принимает три целых числа: два слагаемых и модуль и возвращает сумму двух чисел по модулю. Используя QuickCheck, проверить следующие свойства:
   (a) Сложение по модулю: (addMod x y m) mod m == (x + y) mod m.
   (b) Нейтральный элемент: addMod x 0 m == x mod m.
   (c) Коммутативность: addMod x y m == addMod y x m.
2. **Функция 2:** Реализовать функцию treeDepth :: Tree a -> Int, где Tree a – бинарное дерево, определяемое как: data Tree a = Empty | Node a (Tree a)(Tree a). Используя QuickChech, проверить следующие свойства:
   (a) Глубина пустого дерева равна 0.
   (b) Глубина дерева с одним узлом равна 1.
   (c) Глубина дерева с двумя ветвями равна максимальной глубине среди ветвей плюс 1.
   (d) Добавление узла в дерево не уменьшает его глубину.

Лабораторная работа выполнена на языке Haskell в текстовом редакторе Visual Studio Code 1.95.3.

1 Математическое описание

QuickCheck — это инструмент для автоматической проверки свойств программ. Он использует метод генерации тестов с последующей проверкой утверждений о программе с использованием случайных данных. QuickCheck позволяет описывать свойства функций в виде логических выражений, и затем автоматически проверять их с помощью случайных значений.

В Haskell для работы с QuickCheck есть несколько основных команд и функций, которые позволяют генерировать тесты и проверять свойства:

- **quickCheck:** Команда для запуска проверки свойства. Она принимает функцию с описанием свойства и автоматически генерирует случайные входные данные для тестирования.
- **Property:** Это тип, который представляет собой проверку логического свойства, которое может быть либо истинным, либо ложным, в зависимости от входных данных. Он часто используется с операторами, такими как ==>, чтобы описать свойства, которые должны выполняться при определённых условиях.

Для того чтобы QuickCheck мог генерировать случайные данные для тестов, необходимо определить, как создавать случайные значения для тех типов данных, которые используются в тестируемых функциях. Это делается через класс Arbitrary.

Arbitrary — это класс типов, который предоставляет метод arbitrary, используемый для генерации случайных значений. Для каждого типа, для которого мы хотим генерировать случайные значения, нужно предоставить инстанс этого класса.

2 Особенности реализации

2.1 Функция 1

addMod — это функция, которая выполняет сложение двух целых чисел по модулю третьего числа. Формально, она вычисляет сумму двух чисел x и y, а затем результат делится по модулю z. То есть функция возвращает остаток от деления суммы чисел x и y на число z. Функция представлена на листинге 5.

Листинг 1. Функция вычисления суммы двух чисел по модулю третьего числа

```
addMod :: Int -> Int -> Int -> Int
addMod x y z = (x + y) `mod` z
```

где: x, y, z — целые числа (Int), операция `mod` возвращает остаток от деления $(x+y)$ на z.

Далее были написаны тесты для проверки следующих свойств:

1. **Сложение по модулю:** (addMod x y m) mod m == (x + y) mod m.

Листинг 2. Код теста №1

```
prop_mod :: Int -> Int -> Int -> Property
prop_mod x y m = (m /= 0) ==> (addMod x y m) `mod` m === (x + y) `mod`
    ↪m
```

2. **Нейтральный элемент:** addMod x 0 m == x mod m. Функция представлена на листинге 6.

Листинг 3. Код теста для нейтрального элемента

```
prop_neutral :: Int -> Int -> Property
prop_neutral x m = (m /= 0) ==> addMod x 0 m === x `mod` m
```

3. **Коммутативность:** addMod x y m == addMod y x m. Функция представлена на листинге 7.

Листинг 4. Код теста для коммутативности

```
prop_commutativity :: Int -> Int -> Int -> Property
prop_commutativity x y m = (m /= 0) ==> addMod x y m === addMod y x m
```

2.2 Функция 2

Tree — это рекурсивный тип данных, представляющий бинарное дерево, где каждый узел может иметь два дочерних элемента, а также может быть пустым. Тип данных `Tree` объявляется следующим образом:

Листинг 5. Объявление типа данных Tree

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

Функция **treeDepth** вычисляет глубину дерева. Глубина дерева определяется как максимальное количество узлов от корня до самого глубокого листа. Если дерево пустое (`Empty`), его глубина равна 0. Для узла глубина равна 1 плюс максимальная глубина его левого и правого поддеревьев.

Листинг 6. Функция вычисления глубины дерева

```
treeDepth :: Tree a -> Int
treeDepth Empty = 0
treeDepth (Node _ left right) = 1 + max (treeDepth left) (treeDepth right)
```

Далее были написаны тесты для проверки следующих свойств:

1. **Глубина пустого дерева:** Глубина пустого дерева должна быть равна 0. Тест представлен на листинге 7.

Листинг 7. Тест для проверки глубины пустого дерева

```
prop_emptyTree :: Bool
prop_emptyTree = treeDepth (Empty :: Tree Int) == 0
```

2. **Глубина дерева с одним узлом:** Глубина дерева с одним узлом должна быть равна 1. Тест представлен на листинге 8.

Листинг 8. Тест для проверки глубины дерева с одним узлом

```
prop_singleNodeTree :: Bool
prop_singleNodeTree = treeDepth (Node 1 Empty Empty :: Tree Int) == 1
```

3. **Глубина дерева с двумя ветвями:** Глубина дерева с двумя ветвями должна быть равна максимальной глубине среди ветвей плюс 1. Тест представлен на листинге 9.

Листинг 9. Тест для проверки глубины дерева с двумя ветвями

```
prop_twoBranchTree :: Tree Int -> Tree Int -> Bool
prop_twoBranchTree leftBranch rightBranch =
let tree = Node 1 leftBranch rightBranch
in treeDepth tree == 1 + max (treeDepth leftBranch) (treeDepth
    ↪rightBranch)
```

4. **Добавление узла в дерево не уменьшает его глубину:** При добавлении узла в дерево его глубина не уменьшается. Тест представлен на листинге 10.

Листинг 10. Тест для проверки добавления узла в дерево

```
prop_addNodeInTree :: Tree Int -> Int -> Bool
prop_addNodeInTree tree x =
let newTree = Node x tree Empty
in treeDepth newTree >= treeDepth tree
```

2.3 Генерация случайных данных с помощью Arbitrary

В Haskell библиотека QuickCheck предоставляет тип `Arbitrary`, который используется для автоматической генерации случайных значений для тестируемых типов. Чтобы использовать QuickCheck для тестирования типов данных, необходимо определить экземпляр типа `Arbitrary` для этого типа данных. В случае с деревьями мы определили экземпляр `Arbitrary` для типа данных `Tree a`, который позволяет QuickCheck генерировать случайные деревья для тестирования.

Тип данных `Arbitrary` выглядит следующим образом:

Листинг 11. Интерфейс типа Arbitrary

```
class Arbitrary a where
arbitrary :: Gen a
```

Функция `arbitrary` генерирует случайные значения для типа `a`, и в случае с деревьями мы определяем, как генерировать случайные деревья. В нашем примере используется функция `sized`, которая позволяет задать ограничение на глубину генерируемого дерева.

Вот как выглядит реализация `Arbitrary` для типа `Tree`:

Листинг 12. Определение Arbitrary для типа Tree

```
instance Arbitrary a => Arbitrary (Tree a) where
arbitrary = sized genTree
where
genTree 0 = return Empty
genTree n = frequency
[ (1, return Empty)
, (5, Node <$> arbitrary <*> genTree (n-1) <*> genTree (n-1))
]
```

Здесь: - `sized` позволяет задавать глубину дерева, передавая параметр глубины в функцию `genTree`. - Когда глубина дерева равна 0 (`genTree 0`), генерируется пустое дерево (`Empty`). - Для большей глубины (`genTree n`) с использованием `frequency` выбираются варианты: с вероятностью 1 генерируется пустое дерево, с вероятностью 5 создается узел дерева с рекурсивно генерируемыми поддеревьями.

2.4 Функция main

Функция `main` является точкой входа программы, в которой выполняются тесты для проверки свойств различных функций. В частности, для каждого теста используется функция `quickCheck`, которая автоматически проверяет заданное свойство на случайных данных.

Функция представлена на листинге 13.

Листинг 13. Код функции main

```
main :: IO()
main = do
putStrLn "Сложение по модулю: "
quickCheck prop_mod
putStrLn "Нейтральный элемент: "
quickCheck prop_neutral
putStrLn "Коммутативность: "
quickCheck prop_commutativity
putStrLn "\nГлубина пустого дерева равна 0: "
quickCheck prop_emptyTree
putStrLn "Глубина дерева с одним узлом равна 1: "
quickCheck prop_singleNodeTree
putStrLn "Глубина дерева с двумя ветвями равна максимальной глубине среди в
    ↪етвей плюс 1: "
quickCheckWith stdArgs { maxSuccess = 100, maxSize = 10 }
    ↪prop_twoBranchTree
putStrLn "Добавление узла в дерево не уменьшает его глубину: "
quickCheckWith stdArgs { maxSuccess = 100, maxSize = 10 }
    ↪prop_addNodeInTree
```

3 Результаты работы программы

Результаты запуска программы с помощью команды `stack test` представлены на рисунке 1

```
Сложение по модулю:
+++ OK, passed 100 tests; 11 discarded.
Нейтральный элемент:
+++ OK, passed 100 tests; 12 discarded.
Коммутативность:
+++ OK, passed 100 tests; 15 discarded.

Глубина пустого дерева равна 0:
+++ OK, passed 1 test.
Глубина дерева с одним узлом равна 1:
+++ OK, passed 1 test.
Глубина дерева с двумя ветвями равна максимальной глубине среди ветвей плюс 1:
+++ OK, passed 100 tests.
Добавление узла в дерево не уменьшает его глубину:
+++ OK, passed 100 tests.
```

Рис. 1. Результаты работы программы в консоли.

Тестирование каждого свойства функций `addMod` и `treeDepth` с использованием QuickCheck на наборе 100 случайных входных данных было успешно завершено, все тесты пройдены.

В случае, если функция будет некорретно вычислять значение, тесты свойств будут провалены, как показано на рисунке 2.

```
Глубина пустого дерева равна 0:
*** Failed! Falsified (after 1 test):
Глубина дерева с одним узлом равна 1:
*** Failed! Falsified (after 1 test):
Глубина дерева с двумя ветвями равна максимальной глубине среди ветвей плюс 1:
*** Failed! Falsified (after 1 test):
Empty
Empty
Добавление узла в дерево не уменьшает его глубину:
*** Failed! Falsified (after 1 test):
Empty
0
```

Рис. 2. Тесты провалены.

Для этого мы изменили функцию `treeDepth`, код измененной функции представлен в листинге 14.

Листинг 14. Код измененной функции treeDepth

```haskell
treeDepth :: Tree a -> Int
treeDepth Empty = -1
treeDepth (Node _ left right) = -5 + max (treeDepth left) (treeDepth right)
```

Заключение

В ходе выполнения лабораторной работы была реализована и протестирована на корректность несколько функций в языке программирования Haskell. В частности, была реализована функция сложения по модулю, которая принимает три целых числа и возвращает остаток от деления суммы этих чисел на третье число. Для этой функции были написаны тесты с использованием библиотеки QuickCheck, проверяющие такие свойства, как сложение по модулю, нейтральный элемент и коммутативность операции.

Кроме того, была реализована функция для вычисления глубины бинарного дерева, представленного типом данных `Tree`. Для этой функции также были разработаны тесты с использованием QuickCheck, которые проверяют такие свойства, как глубина пустого дерева, глубина дерева с одним узлом, глубина дерева с двумя ветвями и увеличение глубины при добавлении нового узла в дерево.

В отчете приведены результаты тестирования каждого заданного свойства для обоих функций с использованием библиотеки QuickCheck.

Список литературы

[1] Курт У. Программируй на Haskell / пер. с англ. С. Соловьева. — Москва: ДМК Пресс, 2019. — 384 с.