Лекция 11: тестирование свойств

Функциональное программирование на Haskell

Алексей Романов

6 июня 2019 г.

МИЭТ

Привычный подход к тестированию

- Как обычно мы тестируем написанные функции?

Привычный подход к тестированию

- Как обычно мы тестируем написанные функции?
- Вычисляем функцию на каких-то конкретных аргументах и сравниваем результат с ожидаемым.
- Плюсы:

Привычный подход к тестированию

- Как обычно мы тестируем написанные функции?
- Вычисляем функцию на каких-то конкретных аргументах и сравниваем результат с ожидаемым.
- Плюсы:
  - Тесты легко понять.
  - Можно сделать даже без помощи библиотек.
  - Но при этом библиотеки для написания таких тестов есть практически для всех языков.
  - Для чистых функций (в Haskell почти для всех!) кроме аргументов и результата функции проверять и нечего.

Привычный подход к тестированию

- Как обычно мы тестируем написанные функции?
- Вычисляем функцию на каких-то конкретных аргументах и сравниваем результат с ожидаемым.
- Плюсы:
  - Тесты легко понять.
  - Можно сделать даже без помощи библиотек.
  - Но при этом библиотеки для написания таких тестов есть практически для всех языков.
  - Для чистых функций (в Haskell почти для всех!) кроме аргументов и результата функции проверять и нечего.
- TDD (Test Driven Development)

Привычный подход к тестированию

- Как обычно мы тестируем написанные функции?
- Вычисляем функцию на каких-то конкретных аргументах и сравниваем результат с ожидаемым.
- Плюсы:
  - Тесты легко понять.
  - Можно сделать даже без помощи библиотек.
  - Но при этом библиотеки для написания таких тестов есть практически для всех языков.
  - Для чистых функций (в Haskell почти для всех!) кроме аргументов и результата функции проверять и нечего.
- TDD (Test Driven Development) — сначала тесты, потом реализация.

Недостатки этого подхода

- А какие у этого подхода минусы?

Недостатки этого подхода

- А какие у этого подхода минусы?
- Сколько нужно тестов? Всегда можно добавить ещё.

Недостатки этого подхода

- А какие у этого подхода минусы?
- Сколько нужно тестов? Всегда можно добавить ещё. Разве что если функция на `Bool`...

Недостатки этого подхода

- А какие у этого подхода минусы?
- Сколько нужно тестов? Всегда можно добавить ещё. Разве что если функция на `Bool`...
- Нужно искать пограничные случаи (т.е. те, в которых ошибка более вероятна) самому.
  Хотя есть часто встречающиеся:

Недостатки этого подхода

- А какие у этого подхода минусы?
- Сколько нужно тестов? Всегда можно добавить ещё. Разве что если функция на `Bool`...
- Нужно искать пограничные случаи (т.е. те, в которых ошибка более вероятна) самому.
  Хотя есть часто встречающиеся: `0`, пустой список, `INT_MIN`...

Недостатки этого подхода

- А какие у этого подхода минусы?
- Сколько нужно тестов? Всегда можно добавить ещё. Разве что если функция на `Bool`...
- Нужно искать пограничные случаи (т.е. те, в которых ошибка более вероятна) самому.
  Хотя есть часто встречающиеся: `0`, пустой список, `INT_MIN`...
- Для каких-то аргументов мы можем не знать правильного результата (и не иметь независимого от тестируемой функции способа его найти).

Недостатки этого подхода

- А какие у этого подхода минусы?
- Сколько нужно тестов? Всегда можно добавить ещё. Разве что если функция на `Bool`...
- Нужно искать пограничные случаи (т.е. те, в которых ошибка более вероятна) самому.
  Хотя есть часто встречающиеся: `0`, пустой список, `INT_MIN`...
- Для каких-то аргументов мы можем не знать правильного результата (и не иметь независимого от тестируемой функции способа его найти).
- Есть ситуации, когда программу или функцию в ней намеренно пытаются сломать.

Недостатки этого подхода

- А какие у этого подхода минусы?
- Сколько нужно тестов? Всегда можно добавить ещё. Разве что если функция на `Bool`...
- Нужно искать пограничные случаи (т.е. те, в которых ошибка более вероятна) самому.
  Хотя есть часто встречающиеся: `0`, пустой список, `INT_MIN`...
- Для каких-то аргументов мы можем не знать правильного результата (и не иметь независимого от тестируемой функции способа его найти).
- Есть ситуации, когда программу или функцию в ней намеренно пытаются сломать. Например, злобный преподаватель.

Свойства вместо примеров

- Тестирование свойств (property testing или property-based testing) позволяет избежать этих проблем (и создать новые, как обычно).
- Основная идея: вместо результата на конкретных аргументах мы описываем свойства, которые должны выполняться *для всех аргументов* подходящего типа.

Свойства вместо примеров

- Тестирование свойств (property testing или property-based testing) позволяет избежать этих проблем (и создать новые, как обычно).
- Основная идея: вместо результата на конкретных аргументах мы описываем свойства, которые должны выполняться *для всех аргументов* подходящего типа. Или не совсем для всех, но об этом потом.

Свойства вместо примеров

- Тестирование свойств (property testing или property-based testing) позволяет избежать этих проблем (и создать новые, как обычно).
- Основная идея: вместо результата на конкретных аргументах мы описываем свойства, которые должны выполняться *для всех аргументов* подходящего типа. Или не совсем для всех, но об этом потом.
- Эти свойства проверяем на случайных аргументах (скажем, 100 за раз по умолчанию).

Свойства вместо примеров

- Тестирование свойств (property testing или property-based testing) позволяет избежать этих проблем (и создать новые, как обычно).
- Основная идея: вместо результата на конкретных аргументах мы описываем свойства, которые должны выполняться *для всех аргументов* подходящего типа. Или не совсем для всех, но об этом потом.
- Эти свойства проверяем на случайных аргументах (скажем, 100 за раз по умолчанию).
- В процессе разработки тесты запускаются далеко не один раз, так что каждое свойство будет проверено в сотнях или тысячах случаев.

Простейший пример

- Для Haskell две основных библиотеки: QuickCheck и Hedgehog. Мы используем первую для простоты кода.

```
> import Test.QuickCheck
> quickCheck (\x -> (x :: Integer) * x >= 0)
+++ OK, passed 100 tests.
```

- Проверили, что квадраты `Integer` неотрицательны.

Ещё один пример

- Верно ли, что первый элемент списка всегда равен последнему элементу развёрнутого?

```
> quickCheck (\xs -> head xs === last (reverse
    xs))
```

Ещё один пример

- Верно ли, что первый элемент списка всегда равен последнему элементу развёрнутого?

```
> quickCheck (\xs -> head xs === last (reverse
    xs))
*** Failed! Exception: 'Prelude.head: empty
    list' (after 1 test):
[]
```

- Как исправить?

Ещё один пример

- Верно ли, что первый элемент списка всегда равен последнему элементу развёрнутого?

```
> quickCheck (\xs -> head xs === last (reverse
    xs))
*** Failed! Exception: 'Prelude.head: empty
    list' (after 1 test):
[]
```

- Как исправить? Нужно добавить условие, что список непустой:

```
> quickCheck (\xs -> not (null xs) ==> head xs
    === last (reverse xs))
```

Ещё один пример

- Верно ли, что первый элемент списка всегда равен последнему элементу развёрнутого?

```
> quickCheck (\xs -> head xs === last (reverse
    xs))
*** Failed! Exception: 'Prelude.head: empty
    list' (after 1 test):
[]
```

- Как исправить? Нужно добавить условие, что список непустой:

```
> quickCheck (\xs -> not (null xs) ==> head xs
    === last (reverse xs))
+++ OK, passed 100 tests.
```

Основные типы и функции

- `Gen a`: генератор случайных значений типа `a`.
- `forAll :: (Show a, Testable prop) => Gen a -> (a -> prop) -> Property`: создаёт свойство с использованием заданного генератора.
- `class Arbitrary a where arbitrary :: Gen a`: типы, у которых есть «стандартный генератор».

- `class Testable a`: типы, которые можно протестировать (например `Bool` и `Property`).
- Функции `a -> prop`, где `Arbitrary a`, `Show a`, `Testable prop` — тоже `Testable`.
- `quickCheck, verboseCheck :: Testable prop => prop -> IO ()`: проверяет свойство и выводит результат.

Генераторы значений

- Примеры базовых генераторов:

```
choose :: Random a => (a, a) -> Gen a
elements :: [a] -> Gen a
```

- Из простых генераторов можно строить более сложные, пользуясь тем, что Gen — монада. Например:

```
genPair :: Gen a -> Gen b -> Gen (a, b)
genPair ga gb =
```

Генераторы значений

- Примеры базовых генераторов:

```
choose :: Random a => (a, a) -> Gen a
elements :: [a] -> Gen a
```

- Из простых генераторов можно строить более сложные, пользуясь тем, что Gen — монада. Например:

```
genPair :: Gen a -> Gen b -> Gen (a, b)
genPair ga gb = liftA2 (,) ga gb

genMaybe :: Gen a -> Gen (Maybe a)
genMaybe ga = do is_just <- arbitrary @Bool
                 if is_just
                   then
```

Генераторы значений

- Примеры базовых генераторов:

```
choose :: Random a => (a, a) -> Gen a
elements :: [a] -> Gen a
```

- Из простых генераторов можно строить более сложные, пользуясь тем, что Gen — монада. Например:

```
genPair :: Gen a -> Gen b -> Gen (a, b)
genPair ga gb = liftA2 (,) ga gb

genMaybe :: Gen a -> Gen (Maybe a)
genMaybe ga = do is_just <- arbitrary @Bool
                 if is_just
                   then fmap Just ga
                   else
```

Генераторы значений

- Примеры базовых генераторов:

```
choose :: Random a => (a, a) -> Gen a
elements :: [a] -> Gen a
```

- Из простых генераторов можно строить более сложные, пользуясь тем, что Gen — монада. Например:

```
genPair :: Gen a -> Gen b -> Gen (a, b)
genPair ga gb = liftA2 (,) ga gb

genMaybe :: Gen a -> Gen (Maybe a)
genMaybe ga = do is_just <- arbitrary @Bool
                 if is_just
                   then fmap Just ga
                   else pure Nothing
```

Генераторы значений

- С рекурсивными типами несколько сложнее. Можно попробовать привести пример

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

treeOf :: Gen a -> Gen (Tree a)
treeOf g = oneOf [pure Empty,
                  liftA3 Node g (treeOf g) (treeOf g)]
```

  но это определение плохое: оно даёт очень маленькие деревья (почему?).

- Правильнее:

```
treeOf g = do n <- getSize
              if n == 0
                then pure Empty
                else do
                  k <- choose (0,n)
                  liftA3 Node g (resize k $ treeOf g)
                                (resize (n-k) $ treeOf g)
```

Минимизация контрпримеров

- Найденный сначала контрпример обычно будет довольно большим (хотя начинается генерация с примеров малого размера).
- Поэтому библиотека старается найти и выдать пользователю более простые значения аргументов, дающие ту же ошибку.
- В случае QuickCheck за это уменьшение отвечает функция `shrink :: a -> [a]` в классе `Arbitrary`.
- В Hedgehog, это уменьшение задаётся как часть `Hedgehog.Gen`, что имеет преимущества:
  - Если что-то сгенерировали, точно можно уменьшить.
  - При уменьшении сохраняются инварианты.
  - Подробнее: Integrated vs type based shrinking.

Воспроизводимость

- Если мы нашли контрпример (например, с десятого запуска) и изменили реализацию, нужно проверить, что этот случай исправился.
- В QuickCheck для этого просто запускаем то же свойство с напечатанными аргументами. Можно также добавить обычный тест с ними.
- Конечно, это удобно делать только тогда, когда минимизация более-менее удалась.
- Кроме того, для некоторых типов результат `show x` недостаточен для точного воспроизведения x (придумаете пример?)
- Поэтому в Hedgehog кроме аргументов выдаётся параметр для перезапуска генератора.

Реализация Arbitrary для своих типов

- Для пар:

```
instance (Arbitrary a, Arbitrary b) =>
    Arbitrary (a, b) where
  arbitrary = genPair arbitrary arbitrary
  shrink (x, y) = [(x', y') | x' <- shrink x,
   y' <- shrink y]
```

- Для деревьев:

```
instance Arbitrary a => Arbitrary (Tree a)
    where
  arbitrary = treeOf arbitrary
  shrink Empty = []
  shrink (Node x l r) =
    [Empty, l, r] ++
    [Node x' l' r' |
      (x', l', r') <- shrink (x, l, r)]
```

Часто встречающиеся виды свойств

- Самое тривиальное свойство — функция даёт результат, а не выкидывает исключение.
- Сверка с другой реализацией той же функции.
- При наличии также обратной функции: `encode (decode x) == x`.
- Алгебраические свойства: идемпотентность, коммутативность, ассоциативность и так далее.
- Законы классов типов.

Генераторы функций

- Для тестирования функций высшего порядка нужно уметь генерировать их аргументы, то есть случайные функции. Для этого в QuickCheck есть тип `Fun` и класс `Function`. `Fun a b` представляет функцию `a -> b` как набор пар аргумент-значение и значение по умолчанию (например `{"elephant"->1, "monkey"->1, _->0}`). Экземпляр `Function a` нужен для генерации `Fun a b`.

Свойства функций

- Как пример использования, покажем, что функции `Int -> String` — чистые, то есть дают одинаковый результат при повторном вызове:

```
prop :: Fun Int String -> Int -> Bool
prop (Fn f) x = f x == f x
> quickCheck prop
+++ OK, passed 100 tests.
```

Тестирование законов классов

- Библиотека quickcheck-classes содержит свойства, проверяющие законы для множества классов.
- Например, законы `Eq` определяются функцией `eqLaws :: (Eq a, Arbitrary a, Show a) => Proxy a -> Laws` и могут быть проверены для `Rational` с помощью `lawsCheck (eqLaws (Proxy @Rational))`.
- Для проверки законов `Functor/Applicative/Monad/...` вместо `Proxy` нужно `Proxy1`.
- Для Hedgehog совершенно аналогичная hedgehog-classes (только последняя версия у меня не поставилась).

Ещё раз: Hedgehog или QuickCheck

- Главная разница: в Hedgehog уменьшение интегрировано с генерацией значений, а не определяется типом.
- Нет `Arbitrary`, генераторы контролируются полностью явно.
- Для улучшения воспроизводимости вместе с контрпримером выдаётся seed.
- Улучшенный показ контпримеров с помощью пакета `pretty-show` и функции `diff`.
- Проще работа с монадическими свойствами.
- Поддержка тестирования с состоянием.
- Для генерации функций нужна отдельная библиотека `hedgehog-fn`.
- Параллельная проверка (для ускорения).

Дополнительное чтение

- QuickCheck in Every Language Список библиотек тестирования свойств на разных языках на апрель 2016 (от автора Hypothesis для Python).
- The Design and Use of QuickCheck
- Property testing with Hedgehog
- Как перестать беспокоиться и начать писать тесты на основе свойств
- Property-Based Testing in a Screencast Editor Тестирование свойств в графическом приложении (замечательный пример).
- Using Hypothesis and Pexpect to Test High School Programming Assignments
- Verifying Typeclass Laws in Haskell with QuickCheck