Testing: Introducing Hspec & Quickcheck

Introduction

Testing is a core part of the working programmer's toolkit, and Haskell is no exception. Well-specified types can enable programmers to avoid many obvious and tedious tests that might otherwise be necessary to maintain in untyped programming languages, but there's still a lot of value to be obtained in executable specifications. This chapter will introduce you to testing methods for Haskell

Intents and goals

- The what and why of testing
- Using testing libraries
- Practical examples

Testing Landscape

Unit based testing

Unit testing is a method in which you test the smallest possible parts of an application. These units are individually and independently scrutinised for desired behaviours. Unit testing is better automated but it can also be done manually via a human entering inputs and verifying outputs.

Spec based testing

Spec testing is a somewhat newer version of unit testing. Like unit testing, it tests specific functions independently and asks you to assert that, when given the declared input, the result of the operation will be equal to the desired result. When you run the test, the computer checks that the expected result is equal to the actual result and everyone moves on with their day.

Property based testing

Property testing is a testing method where a subset of a large input space is validated, usually against a property or law some code should abide by

Formal Proofs

Formal methods are system design techniques that use rigorously specified mathematical models to build software and hardware systems. In contrast to other design systems, formal methods use mathematical proof as a complement to system testing in order to ensure correct behaviour.

Conventional Testing

As a starting point, we are going to use the library hspec to demonstrate a test case.

- Let's create an Addition.hs module that includes an addition function
- Let's import the necessary hspec packages and others if necessary
- Let's run the "Mbote Nza" addition with the stack test command

Truth according to Hspec

Example 1

```haskell
module Addition where

import Test.Hspec

main :: IO ()
main = hspec $ do
  describe "Addition" $ do
    it "1 + 1 is greater than 1" $ do
      (1 + 1) > 1 `shouldBe` True
```

```
Prelude> main
Addition
  1 + 1 is greater than 1
```

Example 2

```
main :: IO ()
main = hspec $ do
  describe "Division" $ do
    it "15 divided by 3 is 5" $ do
       dividedBy 15 3 `shouldBe` (5, 2)
    it "22 divided by 5 is 4 remainder 2" $ do
       dividedBy 22 5 `shouldBe` (4, 2)


dividedBy :: Integral a => a -> a -> (a, a)
dividedBy num denom = go num denom 0
  where
    go n d count
      | n < d = (count, n)
      | otherwise =
        go (n - d) d (count + 1)
```

```
Division> main

Division
  15 divided by 3 is 5
  22 divided by 5 is 4 remainder 2

Finished in 0.0012 seconds
2 examples, 0 failures
```

Testing the multiplication

Exercise

Write a function that multiplies two numbers using recursive summation then test it using Hspec. The type of the multiplication function should be:

```
(Eq a, Num a) => a -> a -> a
```

Unit testing limitations

Unit based testing can be coined as example based testing and it comes with a lot of limitations

Revisiting the addition function

Dialogue scene between co-developers

***Me*:**

“We need a function that will add two numbers together, would you mind implementing it?”

Co-worker:

“I just finished implementing the ‘add’ function”

***Me*:**

“Great, have you written unit tests for it?”

Co-worker:

“You want tests as well?” (rolls eyes) “Ok.”

Co-worker:

“I just wrote a test. Look! ‘Given 1 + 2, I expect output is 3’. "

Co-worker:

“So can we call it done now?”

***Me*:**

“Well that’s only one test. How do you know that it doesn’t fail for other inputs?”

Co-worker:

“Ok, let me do another one.”

Co-worker:

“I just wrote another awesome test. ‘Given 2 + 2, I expect output is 4’”

***Me*:**

“Yes, but you’re still only testing for special cases. How do you know that it doesn’t fail for other inputs you haven’t thought of?”

Co-worker:

“You want even more tests?” ***(Tu veux seulement que je teste l’ensemble N???)***

But seriously, my imaginary co-worker’s complaint has some validity: *How many tests are enough?*

So now imagine that rather than being a developer, you are a test engineer who is responsible for testing that the “add” function is implemented correctly.

Unfortunately for you, the implementation is being written by a burned-out, always lazy and often malicious programmer who I will call The Enterprise Developer From Hell, or “EDFH”. (The EDFH has a cousin who you might have heard of).

You are practising test-driven-development, enterprise-style, which means that you write a test, and then the EDFH implements code that passes the test. Unfortunately for you, the EDFH loves to practise malicious compliance, as we will see.

So you start with a test like this (using vanilla Unit / Hspec test style):

```haskell
main :: IO ()
main = hspec $ do
  describe "Addition" $ do
    it "1 add 2 is equal to 3" $ do
      1 `add` 2 `shouldBe` 3
```

The EDFH then implements the add function like this:

```haskell
add :: Integer -> Integer -> Integer
add x y =
 if x==1 && y==2
 then 3
 else 0
```

And your test passes!

When you complain to the EDFH, they say that they are doing **Test Driven Development** properly, and only writing the minimal code that will make the test pass.

Fair enough. So you write another test:

```haskell
main :: IO ()
main = hspec $ do
  describe "Addition" $ do
    it "2 add 2 is equal to 4" $ do
      2 `add` 2 `shouldBe` 4
```

The EDFH then changes the implementation of the add function to this:

```haskell
add x y =
 if x==1 && y==2
 then 3
 else if x==2 && y==2
 then 4
 else 0
```

When you again complain to the EDFH, they point out that this approach is actually a best practice. Apparently it's called "The Transformation Priority Premise".

At this point, you start thinking that the EDFH is being malicious, and that this back-and-forth could go on forever!

Now QuickCheck

hspec does a nice job with spec testing, but we're Haskell users we're never satisfied!! ***hspec can only prove something about particular values***. Can we get assurances that are stronger, something closer to **proofs**? As it happens, we can.

QuickCheck was the first library to offer what is today called property testing. hspec testing is more like what is known as unit testing. The testing of individual units of code whereas ***property testing is done with the assertion of laws or properties***.

Setting up QuickCheck Libraries

…

Simple Example

Maybe ([Either …])

```
function :: Maybe Int -> Maybe Int -> Maybe Int
function  m n = do
    a <- m
    b <- n
    return (a + b)
```

```
import Test.QuickCheck

main :: IO ()
main = hspec $ do

  describe "Addition" $ do
    it "x + 1 is always greater than x" $ do
      property $ \x -> x + 1 > (x :: Int)
```

More on Property based testing

There's just one problem. In order to test the add function we previously implemented, you're making use of the + function. In other words, you are using one implementation to test another.
In some cases that is acceptable (see the use of "test oracles" in a following post), but in general, it's a bad idea to have your tests duplicate the code that you are testing! It's a waste of time and effort, and now you have two implementations to build and keep up to date.
So if you can't test by using +, how can you test?
The answer is to create tests that focus on the ***properties*** of the function – the "requirements". These properties should be things that are true for any correct implementation.

So let's think about what the properties of the add function are.

One way of getting started is to think about how add differs from other similar functions. So for example, what is the difference between add and subtract? Well, for subtract, the order of the parameters makes a difference, while for add it doesn't.
So there's a good property to start with. It doesn't depend on addition itself, but it does eliminate a whole class of incorrect implementations. ***This property is commutativity***

```
main :: IO ()
main = hspec $ do
 describe "Addition is commutative" $ do
   it "add1 2 3 is equal to add1 3 2" $ do
     2 `add1` 3 `shouldBe` 3 `add1` 2
```

The next way is to ask ourselves what would happen if we `add` and then `add` to the result of that?

That leads to the idea that two `add 1`s is the same as one `add 2`. Here's the test:

```
main :: IO ()
main = hspec $ do
 describe "Addition is commutative" $ do
```

```
   it "add1 2 3 is equal to add1 3 2" $ do
     2 `add1` 3 `shouldBe` 3 `add1` 2
   it "two add1 1 is the same as add1 2"
     (3 `add1` 1) `add` 1 `shouldBe` 3 `add1` 2
```

What happens when you add zero to a number? You always get the same number back.

```
main :: IO ()
main = hspec $ do
 describe "Addition is commutative" $ do
   it "add1 2 3 is equal to add1 3 2" $ do
     2 `add1` 3 `shouldBe` 3 `add1` 2







   it "two add1 1 is the same as add1 2"
     (3 `add1` 1) `add` 1 `shouldBe` 3 `add1` 2



   it "return the 3 as result"
     0 `add1` 3 `shouldBe` 3
   it "return the 7 as result"
     0 `add1` 7 `shouldBe` 7
```

`One last one is associativity Grouping does not matter`

```
main :: IO ()
main = hspec $ do
 describe "Addition is commutative" $ do
   it "add1 2 3 is equal to add1 3 2" $ do
     2 `add1` 3 `shouldBe` 3 `add1` 2
   it "two add1 1 is the same as add1 2"
     (3 `add1` 1) `add` 1 `shouldBe` 3 `add1` 2
   it "return the 3 as result"
```

```
    0 `add1` 3 `shouldBe` 3
  it "return the 7 as result"
    0 `add1` 7 `shouldBe` 7



  it "meets the associativity property"
    (2 `add1` 3) `add1` 4 `shouldBe` 2 `add1` (3 `add1` 4)
```

So now we have a set of properties that can be used to test any implementation of `add`, and that force the EDFH to create a correct implementation:

Refactoring the common code

There's quite a bit of duplicated code in these three tests. Let's do some refactoring.

First, we'll write a function called `propertyCheck` that does the work of generating

With this in place, we can redefine one of the tests by pulling out the property into a separate function, like this:

Coming back after Arbitrary instances

- commutativeProperty x y = …
  addDoesNotDependOnParameterOrder () =
          propertyCheck commutativeProperty
- associativityProperty x y = …
  addDoesNotDependOnGroupingOrder () =
          propertyCheck associativityProperty
- add1TwiceIsAdd2Property x = …
  addOneTwiceIsSameAsAddTwo () =
          propertyCheck add1TwiceIsAdd2Property

- identityProperty x =
  addZeroIsSameAsDoingNothing () = ...

Reviewing what we have done so far

We have defined a set of properties that any implementation of add should satisfy:

- The parameter order doesn't matter ("commutative" property)
- Addition is associative: the grouping order does not matter
- Doing add twice with 1 is the same as doing add once with 2
- Adding zero does nothing ("identity" property)

What's nice about these properties is that they work with *all* inputs, not just special magic numbers. But more importantly, they show us the core essence of addition.

In fact, you can take this approach to the logical conclusion and actually *define* addition as anything that has these properties.

This is exactly what mathematicians do. If you look up addition on Wikipedia, you'll see that it is defined entirely in terms of commutativity, associativity, identity, and so on.

Specification by properties

- A collection of properties like this can be considered a *specification*.
- Historically, unit tests, as well as being functional tests, have been used as a sort of specification as well. But an approach to specification using properties instead of tests with "magic" data is an alternative which I think is often shorter and less ambiguous.
- You might be thinking that only mathematical kinds of functions can be specified this way, but in future testing modules, we'll see how this approach can be used to test web services and databases too.
- Of course, not every business requirement can be expressed as properties like this, and we must not neglect the social component of software development. Specification by example and domain driven design can play a valuable role when working with non-technical customers.

You also might be thinking that designing all these properties is a lot of work – and you'd be right! It is the hardest part. In a follow-up post, I'll present some tips for coming up with properties which might reduce the effort somewhat.

But even with the extra effort involved upfront (the technical term for this activity is called "thinking about the problem", by the way) the overall time saved by having automated tests and unambiguous specifications will more than pay for the upfront cost later.

In fact, the arguments that are used to promote the benefits of unit testing can equally well be applied to property-based testing! So if a TDD fan tells you that they don't have the time to come up with property-based tests, then they might not be looking at the big picture.

Arbitrary Instances

One of the more important parts of QuickCheck is learning to write instances of the Arbitrary typeclass for your datatypes. It's a somewhat unfortunate but still necessary convenience for your code to integrate cleanly with QuickCheck code. It's initially a bit confusing for beginners because it compacts a few different concepts and solutions to problems into a single typeclass

Babby's First Arbitrary

```haskell
module Main where

import Test.QuickCheck

data Trivial = Trivial deriving (Eq, Show)

trivialGen :: Gen Trivial
trivialGen = return Trivial



instance Arbitrary Trivial where
  arbitrary = trivialGen

main :: IO ()
main = do
  sample trivialGen
```

Let's take a sample:

```
Prelude> sample trivialGen

Trivial
```

```
Trivial
Trivial
Trivial
Trivial
Trivial
Trivial
Trivial
Trivial
Trivial
Trivial
```

Identity Arbitrary

```
data Identity a = Identity a deriving (Eq, Show)

identityGen :: Arbitrary a => Gen (Identity a)
identityGen = do
  a <- arbitrary
  return (Identity a)

instance Arbitrary a => Arbitrary (Identity a) where
  arbitrary = identityGen

identityGenInt :: Gen (Identity Int)
identityGenInt = identityGen
```

Let’s test it:

```
Prelude> sample identityGenInt
Identity 0
Identity (-1)
Identity 2
Identity 4
Identity (-3)
Identity 5
Identity 3
Identity (-1)
Identity 12
Identity 16
Identity 0
```

Arbitrary Products

```haskell
data Pair a b =
  Pair a b
  deriving (Eq, Show)

pairGen :: (Arbitrary a, Arbitrary b) => Gen (Pair a b)
pairGen = do
  x <- arbitrary
  y <- arbitrary
  return (Pair x y)

instance (Arbitrary a, Arbitrary b) => Arbitrary (Pair a b) where
  arbitrary = pairGen

pairGenIntString :: Gen (Pair Int String)
pairGenIntString = pairGen
```

Testing …

```
Pair 0 ""
Pair (-2) ""
Pair (-3) "26"
Pair (-5) "B\NUL\143:\254\SO"
Pair (-6) "\184*\239\DC4"
Pair 5 "\238\213=J\NAK!"
Pair 6 "Pv$y"
Pair (-10) "G|J^"
Pair 16 "R"
Pair (-7) "("
Pair 19 "i\ETX]\182\ENQ"
```

Greater than the sum of its parts

Writing Arbitrary instances for sum types is a bit more interesting still. First, make sure the following is included in your imports:

```
import Test.QuickCheck.Gen (oneof)
```

```
...
data Sum a b =
   First a
 | Second b
 deriving (Eq, Show)

-- equal odds for each



sumGenEqual :: (Arbitrary a,Arbitrary b) => Gen (Sum a b)
sumGenEqual = do
 x <- arbitrary
 y <- arbitrary
 oneof [return $ First x, return $ Second y]


sumGenCharInt :: Gen (Sum Char Int)
sumGenCharInt = sumGenEqual
```

Testing we get:

```
Prelude> sample sumGenCharIntCHAPTER
First 'P'
First '\227'
First '\238'
First '.'
Second (-3)
First '\132'
Second (-12)
Second (-12)
First '\186'
Second (-11)
First '\v'
```

Where sum types get even more interesting is that you can choose a different weighting of probabilities than an equal distribution. Consider this snippet of the Maybe Arbitrary instance from the QuickCheck library:

```haskell
sumGenFirstPls :: (Arbitrary a, Arbitrary b) => Gen (Sum a b)
sumGenFirstPls = do
  a <- arbitrary
  b <- arbitrary
  frequency [(10, return $ First a),(1, return $ Second b)]

sumGenCharIntFirst :: Gen (Sum Char Int)
sumGenCharIntFirst = sumGenFirstPls
```

Testing we get:

```
First '\208'
First '\242'
First '\159'
First 'v'
First '\159'
First '\232'
First '3'
First 'l'
Second (-16)
First 'x'
First 'Y'
```

CoArbitrary (To research… as homework)

Standalone QuickCheck

```haskell
mainQuickCheckTest :: IO ()
mainQuickCheckTest = do
 let ma = monoidAssoc
     mli = monoidLeftIdentity
     mri = monoidRightIdentity
     fui = functorIdentity
     fui' = functorIdentity
     fuc = functorCompose (+1) (*2)
     fuc' x = fuc (x :: [Int])
     fucNew = functorCompose (+1) (*2)
     fucNew' x = fucNew (x :: MyMaybe Int)
     foi = identityCompose (*2)
     foi' x = foi (x :: Int)
     idc = (identityCompose' :: IntFC)
 quickCheck (ma :: String -> String -> String -> Bool)
 quickCheck (mli :: String -> Bool)
 quickCheck (mri :: String -> Bool)
 quickCheck (fui :: [Int] -> Bool)
 quickCheck (fui' :: MyMaybe Int -> Bool)
 quickCheck fuc'
 quickCheck foi'
 quickCheck idc
 quickCheck fucNew'
```

```haskell
monoidAssoc :: (Eq m, Monoid m) => m -> m -> m -> Bool
monoidAssoc a b c =
(a <> (b <> c)) == ((a <> b) <> c)

monoidLeftIdentity :: (Eq m, Monoid m) => m -> Bool
monoidLeftIdentity a = (mempty <> a) == a

monoidRightIdentity :: (Eq m, Monoid m) => m -> Bool
monoidRightIdentity a = (a <> mempty) == a

functorIdentity :: (Functor f, Eq (f a)) => f a -> Bool
functorIdentity x =
  fmap id x == x

functorCompose :: (Eq (f c), Functor f) => (a -> b) -> (b -> c) -> f a -> Bool
functorCompose f g x =
(fmap g (fmap f x)) == (fmap (g . f) x)

identityCompose :: Eq b => (a -> b) -> a -> Bool
identityCompose f x =
 (id (f x)) == (f (id x))


identityCompose' :: Eq b => Fun a b -> a -> Bool
```

```haskell
identityCompose' (Fun _ f) x =
 (id (f x)) == (f (id x))

type IntToInt = Fun Int Int
type IntFC = IntToInt -> Int -> Bool
```

Application Examples

Resole all exercises from the course book : ***Haskell Programming from First Principles***

Common Patterns Laws Testing

Monoids

Laws

Left identity

```haskell
mappend mempty x = x
```

```haskell
monoidRightIdentity :: (Eq m, Monoid m) => m -> Bool
monoidRightIdentity a = (a <> mempty) == a
```

Right identity

```haskell
mappend x mempty = x
```

```haskell
monoidRightIdentity :: (Eq m, Monoid m) => m -> Bool
monoidRightIdentity a = (a <> mempty) == a
```

Associativity

```
asc :: Eq a => (a -> a -> a) -> a -> a -> a -> Bool
asc (<>) a b c =
  a <> (b <> c) == (a <> b) <> c
```

```
mappend x (mappend y z) =
  mappend (mappend x y) z

mconcat = foldr mappend mempty
```

```
import Data.Monoid
import Test.QuickCheck

monoidAssoc :: (Eq m, Monoid m) => m -> m -> m -> Bool
monoidAssoc a b c =
  (a <> (b <> c)) == ((a <> b) <> c)
```

Testing through the GHC compiler …

```
Prelude> type S = String
Prelude> type B = Bool
Prelude> quickCheck (monoidAssoc :: S -> S -> S -> B)
+++ OK, passed 100 tests.
```

An Invalid monoid

```
import Test.QuickCheck
```

```haskell
data Bull =
  Fools
| Twoo
deriving (Eq, Show)

instance Arbitrary Bull where
  arbitrary =
    frequency [ (1, return Fools), (1, return Twoo) ]

instance Monoid Bull where
  mempty = Fools
  mappend _ _ = Fools

type BullMappend =
  Bull -> Bull -> Bull -> Bool

main :: IO ()
main = do
  let ma = monoidAssoc
      mli = monoidLeftIdentity
      mlr = monoidRightIdentity
  quickCheck (ma :: BullMappend)
  quickCheck (mli :: Bull -> Bool)
  quickCheck (mlr :: Bull -> Bool)
```

Result …

```
Prelude> main
+++ OK, passed 100 tests.
*** Failed! Falsifiable (after 1 test):
Twoo
*** Failed! Falsifiable (after 1 test):
Twoo
```

Functors

Laws

Identity

```
fmap id = id
```

```
functorIdentity :: (Functor f, Eq (f a)) => f a -> Bool
functorIdentity f = fmap id f == f
```

Composition

```
fmap (f . g) = (fmap f) . (fmap g)
```

```
functorCompose :: (Eq (f c), Functor f) => (a -> b) -> (b -> c) -> f a
-> Bool
functorCompose f g x = (fmap g (fmap f x)) == (fmap (g . f) x)
```

Testing in the terminal

```

Prelude> :{
*Main| let f :: [Int] -> Bool
*Main| f x = functorIdentity x
*Main| :}

Prelude> quickCheck f
+++ OK, passed 100 tests.

Prelude> let c = functorCompose (+1) (*2)
Prelude> let li x = c (x :: [Int])
Prelude> quickCheck li
+++ OK, passed 100 tests.
```

Generating functions

```
{-# LANGUAGE ViewPatterns #-}

import Test.QuickCheck
import Test.QuickCheck.Function

functorCompose' :: (Eq (f c), Functor f) =>
f a
-> Fun a b
-> Fun b c
-> Bool
functorCompose' x (Fun _ f) (Fun _ g) =
  (fmap (g . f) x) == (fmap g . fmap f $ x)
```

Testing …

```
Prelude> type IntToInt = Fun Int Int
Prelude> :{
*Main| type IntFC =
*Main|         [Int]
*Main|      -> IntToInt
*Main|      -> IntToInt
*Main|      -> Bool
*Main| :}

Prelude> let fc' = functorCompose'
Prelude> quickCheck (fc' :: IntFC)
+++ OK, passed 100 tests.
```

Unvalid functor

Given

```
data WhoCares a =
   ItDoesnt
 | Matter a
 | WhatThisIsCalled
 deriving (Eq, Show)


instance Functor WhoCares where
  fmap _ ItDoesnt = WhatThisIsCalled
  fmap _ WhatThisIsCalled = ItDoesnt
  fmap f (Matter a) = Matter (f a)
```

```
data CountingBad a =
  Heisenberg Int a
  deriving (Eq, Show)


instance Functor CountingBad where
  fmap f (Heisenberg n a) =
    Heisenberg (n+1) (f a)
```

Catching it with eyes …

```
Prelude> fmap id ItDoesnt
WhatThisIsCalled

Prelude> fmap id WhatThisIsCalled
ItDoesnt

Prelude> fmap id ItDoesnt == id ItDoesnt
False
```

```
Prelude> :{
*Main| fmap id WhatThisIsCalled ==
*Main|      id WhatThisIsCalled
*Main| :}
False
```

```
Prelude> let u = "Uncle"
Prelude> let oneWhoKnocks = Heisenberg 0 u

Prelude> fmap (++" Jesse") oneWhoKnocks
Heisenberg 1 "Uncle Jesse"

Prelude> let f = ((++" Jesse").(++" lol"))

Prelude> fmap f oneWhoKnocks
Heisenberg 1 "Uncle lol Jesse"
```

So far it seems OK, but what if we compose the two concatenation functions separately?

```
Prelude> let j = (++ " Jesse")
Prelude> let l = (++ " lol")
Prelude> fmap j . fmap l $ oneWhoKnocks
Heisenberg 2 "Uncle lol Jesse"
```

Or let's make it more like a law

```
Prelude> let f = (++" Jesse")
Prelude> let g = (++" lol")
Prelude> fmap (f . g) oneWhoKnocks
Heisenberg 1 "Uncle lol Jesse"
Prelude> fmap f . fmap g $ oneWhoKnocks
Heisenberg 2 "Uncle lol Jesse"
```

Exercise

Unsig QuiCheck, prove that at least one the functor’s property we defined abose is broken

Applicatives ***(Homework)***

Prove all applicatives laws with quickcheck for common applicatives

- Maybe
- [ ]
- Either
- Writer
- State
- …etc

Laws

Identity

```
pure id <*> v = v
```

Composition

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

Homomorphism

```
pure f <*> pure x = pure (f x)
```

Interchange

```
u <*> pure y = pure ($ y) <*> u
```

Monads ***(Homework)***

Prove all monads laws with quickcheck for common monads

- Maybe
- [ ]
- Either
- Writer
- State
- …etc

Laws

Note

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Left identity

```
return x >>= f = f x
```

Right identity

```
m >>= return = m
```

Associativity

```
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

Note: We encourage you to refer to the book Haskell Programming from First Principles and Stackage for a more in depth learning of quickcheck.