Если вы видите что-то необычное, просто сообщите мне.

TDD ? ??????? ???????????

Сколько раз вы встречали зависимость от ошибок в вашем коде? Это может быть очень обидно для разработчика програмного обеспечеия. Вы отправили код в уверенности, что он отлично работает. Но теперь оказалось, что сломано, что-то другое. Даже хуже, вы обнаружили, что несмотря на то что ваш код правилно работает, он делает это очень медленно. Ваша система начинает ломаться при увеличении нагрузки, оставляя плохое впечаление пользователям.

Лучший способ избежать этих проблем это иметь автоматический код, который проверят состояние и производительность ваших программ. В этой части над тестированием в Haskell, мы посмотрим, что библиотеки могут использовать для тестирования и профилирования нашего кода. Первая статья пройдет по общей идее стоящей за TDD, и некоторыми базовыми библиотеками The best way to avoid these issues is to have automated code that verifies test conditions and the performance of your program. In this series on Testing with Haskell, we'll see what libraries we can use to test and profile our code. This first part goes over the general ideas behind test driven development (TDD) and some of the basic libraries we can use to make it work in Haskell. We'll also quickly examine why Haskell is a good fit for TDD.

If you're already familiar with libraries like HUnit and HSpec, you can move onto part 2 of this series, where we discuss how to identify performance issues using profiling.

To use testing properly, you'll need to have some understanding of how we organize projects in Haskell. I recommend you learn how to use Stack to organize your Haskell code. Learn how by taking our free Stack mini-course!

You can follow along with this code on the companion Github Repository for this series! In a few spots we'll reference specific files you can look at, so keep your eyes peeled!

FUNCTIONAL TESTING ADVANTAGES

Testing works best when we are testing specific functions. We pass input, we get output, and we expect the output to match our expectations. In Haskell, this is a approach is a natural fit. Functions are first class citizens. And our programs are largely defined by the composition of functions. Thus our code is by default broken down into our testable units.

Compare this to an object oriented language, like Java. We can test the static methods of a class easily enough. These often aren't so different from pure functions. But now consider calling a method on an object, especially a void method. Since the method has no return value, its effects are all internal. And often, we will have no way of checking the internal effects, since the fields could be private.

We'll also likely want to try checking certain edge cases. But this might involve constructing objects with arbitrary state. Again, we'll run into difficulties with private fields.

In Haskell, all our functions have return values, rather than depending on effects. This makes it easy for us to check their true results. Pure functions also give us another big win. Our functions generally have no side effects and do not depend on global state. Thus we don't have to worry about as many pathological cases that could impact our system.

TEST DRIVEN DEVELOPMENT

So now that we know why we're somewhat confident about our testing, let's explore the process of writing tests. The first step is to define the public API for a particular module. To do this, we define a particular function we're going to expose, and the types that it will take as input as output. Then we can stub it out as undefined, as suggested in this article on Compile Driven Learning. This makes it so that our code that calls it will still compile.

Now the great temptation for much all developers is to jump in and write the function. After all, it's a new function, and you should be excited about it!

But you'll be much better off in the long run if you first take the time to define your test cases. You should first define specific sets of inputs to your function. Then you should match those with the expected output of those parameters. We'll go over the details of this in the next section. Then you'll write your tests in the test suite, and you should be able to compile and run the tests. Since your function is still undefined, they'll all fail. But now you can implement the function incrementally.

Your next goal is to get the function to run to completion. Whenever you find a value you aren't sure how to fill in, try to come up with a base value. Once it runs to completion, the tests will tell you about incorrect values, instead of errors. Then you can gradually get more and more things right. Perhaps some of your tests will check out, but you missed a particular corner case. The tests will let you know about it.

WRITING OUR TEST SUITE

Suppose to start out, we're writing a function that will take three inputs. It should multiply the first two, and subtract the third. We'll start out by making it undefined. You can see this function in this module in the "library" of our Haskell project:

```
simpleMathFunction :: Int -> Int -> Int -> Int
simpleMathFunction a b c = undefined
```

Now let's write a test suite that will evaluate this function! To do this we'll go into the .cabal file for our project and add a test-suite section that looks like this:

```
test-suite unit-test
  type: exitcode-stdio-1.0
  main-is: UnitTest.hs
  other-modules:
      Paths_Testing
  hs-source-dirs:
      test
  ghc-options: -threaded -rtsopts -with-rtsopts=-N
  build-depends:
      Testing
    , base >=4.7 && <5
```

```
    , tasty
    , tasty-hunit
  default-language: Haskell2010
```

A test suite is like an executable. So it has a "Main" module specified by the main-is file, and you should specify the directory it lives in. Many of the other properties are pretty standardized. But the build-depends section will change depending on the test library you decide to use. In our case, we're going to test our code using the HUnit library combined with the Tasty framework.

USING HUNIT

We start out our test suite the same way we start out an executable, by creating a main function of type IO ():

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

main :: IO ()
main = ...
```

Most testing libraries have some kind of a "default" main function you can use that will provide most of their functionality. In the case of HUnit, we'll use defaultMain and then provide a TestTree expression:

```
main :: IO ()
main = ...

simpleMathTests :: TestTree
simpleMathTests = ...
```

We construct a "tree" in two ways. The first is to use an individual case with testCase. This function takes name to identify the case, and then a "predicate assertion".

```
simpleMathTests :: TestTree
simpleMathTests = testCase "Small Numbers" $
```

```
  ... -- (predicate assertion)
```

Ultimately, an assertion is just an IO action. But there are some special combinators we can use to make statements about the function of our code. The most common of these in HUnit are (@?=) and (@=?). These take two expressions and assert that they are equal. One of these should be the "actual" value we get from running our code, and the other should be the "expected" value. Here's our complete test case:

```
simpleMathTests :: TestTree
simpleMathTests = testCase "Small Numbers" $
  simpleMathFunction 3 4 5 @?= 7
```

The @=? operator works the same way, except you should reverse the "actual" and "expected" sides.

The other way to build a TestTree is to use testGroup. This simply takes a name for this layer of the tree, and then a list of TestTree elements. We can then use testCase for those specific elements.

```
simpleMathTests :: TestTree
simpleMathTests = testGroup "Simple Math Tests"
  [ testCase "Small Numbers" $
      simpleMathFunction 3 4 5 @?= 7
  ]
```

If you go to this file in the repository, you can add additional test cases to this list and run them!

RUNNING OUR TESTS

Our basic test suite is now complete! We can run this suite from our project directory by using the following command:

```
stack build Testing:test:unit-test
```

We can also use stack test to run all the different test suites we have. With our undefined function, we'll get this output:

```
Simple Math Tests
  Small Numbers:  FAIL
    Exception: Prelude.undefined
```

So as expected, our test cases fail, so we know how we can go about improving our code. So let's implement this function:

```
simpleMathFunction :: Int -> Int -> Int -> Int
simpleMathFunction a b c = a * b - c
```

And now everything succeeds!

```
Simple Math Tests
  Small Numbers:  OK


All 1 test passed (0.00s)
```

BEHAVIOR DRIVEN DEVELOPMENT

As you work on bigger projects, you'll find you aren't just interacting with other engineers on your team. There are often less technical stakeholders like project managers and QA testers. These folks are less concerned with the internal details of the code, but are focused more on its broader behavior. In these cases, you may want to adopt "behavior driven development." This is like test driven development, but with a different flavor. In this framework, you describe your code and its expected effects via a set of behaviors. Ideally, these are abstract enough that less technical people can understand them.

You as the engineer then want to be able to translate these behaviors into code. Luckily, Haskell is an immensely expressive language. You can often define your functions in such a way that they can almost read like English.

HSPEC

In Haskell, you can implement behavior driven development with the Hspec library. With this library, you describe your functions in a particularly expressive way. All your test specifications will belong to a Spec monad.

In this monad, you can use composable functions to describe the test cases. You will generally begin a description of a test case with the "describe" function. This takes a string describing the general overview of the test case.

```
simpleMathSpec :: Spec
simpleMathSpec = describe "Tests of our simple math function" $ do
  ...
```

You can then modify it by adding a different "context" for each individual case. The context function also takes a string. However, the idiomatic usage of context is that your string should begin with the words "when" or "with".

```
simpleMathSpec :: Spec
simpleMathSpec = describe "Tests of our simple math function" $ do
  context "when the numbers are small" $
    ...
  context "when the numbers are big" $
    ...
```

Now you'll describe each the actual test cases. You'll use the function "it", and then a comparison. The combinators in the Hspec framework are functions with descriptive names like shouldBe. So your case will start with a sentence-like description and context of the case. The the case finishes "it should have a certain result": x "should be" y. Here's what it looks like in practice:

```
simpleMathSpec :: Spec
simpleMathSpec = describe "Tests of our simple math function" $ do
  context "when the numbers are small" $
    it "Should match the our expected value" $
      simpleMathFunction 3 4 5 `shouldBe` 7
  context "when the numbers are big" $
    it "Should match the our expected value" $
      simpleMathFunction 22 12 64 `shouldBe` 200
```

It's also possible to omit the context completely:

```
simpleMathSpec :: Spec
simpleMathSpec = describe "Tests of our simple math function" $ do
  it "Should match the our expected value" $
    simpleMathFunction 3 4 5 `shouldBe` 7
  it "Should match the our expected value" $
    simpleMathFunction 22 12 64 `shouldBe` 200
```

Now to incorporate this into your main function, all you need to do is use hspec together with your Spec!

```
main :: IO ()
main = hspec simpleMathSpec
```

Note that Spec is a monad, so multiple tests are combined with "do" syntax. You can explore this library more and try writing your own test cases in this file in the repository!

At the end, you'll get neatly formatted output with descriptions of the different test cases. By writing expressive function names and adding your own combinators, you can make your test code even more self documenting.

```
Tests of our simple math function
  when the numbers are small
    Should match the our expected value
  when the numbers are big
    Should match the our expected value

Finished in 0.0002 seconds
2 examples, 0 failures
```

CONCLUSION

This concludes our introduction to testing in Haskell. We went through a brief description of the general practices of test-driven development. We saw why it's even more powerful in a functional, typed language like Haskell. We went over some of the basic testing mechanisms you'll find in the HUnit library. We then described the process of "behavior driven development", and how it differs from normal TDD. We concluded by showing how the HSpec library brings BDD to life in Haskell.

But testing correctness is only half the story! We also need to be sure that our code is performant enough. In part 2 of this series, we'll discuss how we can use the Criterion library to identify performance issues in our system.

If you want to see TDD in action and learn about a cool functional paradigm along the way, you should check out our Recursion Workbook. It has 10 practice problems complete with tests, so you can walk through the process of incrementally improving your code and finally seeing the tests pass!

If you want to learn the basics of writing your own test suites, you need to understand how Haskell code is organized! Take our quick and free Stack mini-course to learn how to use the Stack tool for this!

Revision #2
Created 2022-03-11 05:48:28 UTC by gasick
Updated 2022-10-09 17:17:24 UTC by gasick