Testing Basics

We discuss here some basic ideas around testing, that you will further work on in the lab and in future labs and projects.

Automated Tests

When we refer to tests, we typically mean automated tests. Those tests fall into various categories:

Unit Tests Unit tests test a single tiny individual component of your application. You typically want to have unit tests for every bit of your code that is part of your application's *interface*.

You should avoid testing for things that are too implementation-dependent.

Unit tests are a crucial part of the refactoring process, whose goal is to rearrange and rewrite sections of your code. A solid suite of unit tests can allow you to do this freely without worrying about breaking code. "Your tests will catch that". And version control allows you to recover if you've messed things up too much.

Integration Tests Integration tests test bigger parts of your application, making sure that different parts come together naturally.

Timing Tests Timing tests are used in algorithm implementations to assess the efficiency of the algorithms.

They can also be used to try to find bottlenecks in your application, though some of the browser profiling tools might be better.

Deployment Tests Deployment tests are meant to ensure that your application performs well on various browsers / deployment environments. Hard to do.

We will focus on unit tests for now.

Test-Driven Development

In Test-Driven Development, you typically would follow these steps:

- Decide on a small piece of functionality you want to add.
- Make a GitHub issue about it.
  - If you prefer, you can create one bigger more "logical" issue, and create a "task list" in it, following the example at this blog post[1]. Then check those items off as you implement them.

[1] https://github.com/blog/1375%0A-task-lists-in-gfm-issues-pulls-comments

- Write a test for the code you want to introduce.
- Run your tests, and watch this new test fail. This makes us more certain that the test does indeed detect the feature we want to add.
- Optionally, make a git commit of the test, using "ref #..." to reference the issue you created.
  - This is a bit of a style decision, whether to commit the tests separately or whether to do one commit containing both test and new code.
- Write a minimal set of code that would make the test pass.
- Check that all your tests pass.
- Make a commit, using "ref #..." to reference the issue you created. Say "close #..." if it was a "single-problem issue".
  - This gives you a safe backup point to revert to.
- Consider any refactoring that you might want to do to clean things up.
- Do the refactoring, and make sure your tests still all pass.
- Commit (optionally creating an issue first to explain what the refactoring was about).

This is some of the general theory behind testing and test-driven development. We will now look at testing in Haskell.

Testing in Haskell

Haskell offers a number of different testing systems that you can use. In this lab we will focus on two:

HSpec HSpec[2] is a standard "testing specifications" framework. It aims to express tests in as readable a form as possible.

QuickCheck QuickCheck[3] is an awesome library that allows us to test "properties" and to automatically generate many test values. It can be used on its own in the `ghci` interpreter, or as part of a testing framework like HSpec.

[2] https://hspec.github.io/
[3] http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html

Testing with QuickCheck

We start by taking a look at QuickCheck. QuickCheck operates by testing "properties". A **property** is simply a function that returns a boolean (we have been calling these "predicates"). In order for QuickCheck to do its work, it needs to be able to generate various inputs. QuickCheck already knows about many standard inputs, and you can provide it with "generators" for your custom input types.

Let us take a look at an example in the interpreter. We start by loading the QuickCheck module into our module:

```
import Test.QuickCheck
```

In order to use QuickCheck we must define "properties". These are meant to be functions that should be true about **any** input you give them. QuickCheck can't typically check all inputs, but it will check at random 100 inputs.

```
-- This is of course a "wrong" property, as it is not always true.
prop_allNumbersAreLessThan10 :: Int -> Bool
prop_allNumbersAreLessThan10 x =  x < 10
```

It is customary to start the name of these properties via prop.

We can now load our module in gchi and test this property:

```
quickCheck prop_allNumbersAreLessThan10
```

You get back an answer that looks like this:

```
*** Failed! Falsifiable (after 15 tests and 1 shrink):
10
```

So QuickCheck will tell you that the test failed after trying 15 input values. You can try verboseCheck to see the actual values tried:

```
verboseCheck prop_allNumbersAreLessThan10
```

Note that these tests are not perfect: They only test a small number of inputs. For example if we were using 100 instead of 10, we would have gotten back the answer that everything is OK! The test never checked numbers bigger than 100.

Generators We can help QuickCheck along by providing "custom generators" for it. Generators have type Gen a. There are a number of built-in functions that allow us to create new generators:

```
-- Random element between the two given values (inclusive)
choose :: Random a => (a, a) -> Gen a
-- Randomly chooses one of the generators from the list
oneOf :: [Gen a] -> Gen a
-- Randomly chooses a generator from the list, using the "weights" to determine frequency
frequency :: [(Int, Gen a)] -> Gen a
-- Randomly chooses an element from the list
elements :: [a] -> Gen a
-- Generates a value that passes the predicate
suchThat :: Gen a -> (a -> Bool) -> Gen a
```

```
-- Generates a value if the function returns a Just v
suchThatMap :: Gen a -> (a -> Maybe b) -> Gen b
-- Generates a list of elements of random length
listOf :: Gen a -> Gen [a]
listOf1 :: Gen a -> Gen [a]              -- Guaranteed non-empty
vectorOf :: Int -> Gen a -> Gen [a]      -- Specified length
shuffle :: [a] -> Gen [a]                -- Random permutation of the list
-- Arbitrary t means that quickCheck knows how to generate values of type t
arbitrary :: Arbitrary a => Gen a
vector :: Arbitrary a => Int -> Gen [a]
orderedList :: (Ord a, Arbitrary a) => Gen [a]
```

You can use a specific generator as follows:

```
genSmallNumbers :: Gen Int
genSmallNumbers = choose (1, 9)

-- Note the different type
prop_allNumbersAreLessThan10 :: Property
prop_allNumbersAreLessThan10 = forAll genSmallNumbers $ \x -> x < 10
```

In this instance we tell it to use our genSmallNumbers generator to produces the x. That generator in turn only produces numbers between 1 and 9. Thefore QuickCheck will actually succeed in this instance.

Lets try to write a generator that returns powers of a certain base (one easy way to get somewhat large numbers). We will use the do syntax for this, which you are probably not too familiar with, and we will discuss it more later.

```
genPowers :: Int -> Gen Int
genPowers base = do
    expo <- choose (1, 40)
    return (base ^ (expo :: Int))

prop_allNumbersAreLessThan100 :: Property
prop_allNumbersAreLessThan100 = forAll (genPowers 2) $ \x -> x < 100
```

Calling quickCheck on prop_allNumbersAreLessThan100 will now actually fail (as it should) after just a few tries.

There are many more features that QuickCheck provides. You'll want to look at its manual for more.

Specification Testing with HSpec

HSpec is used in order to write more systematic tests, using QuickCheck or direct cases along the way, and produce suitable input. With HSpec you would be creating a separate test file with a suitable main method. We will discuss this kind of testing setup later, after we discuss IO in more detail.