Property-Based Testing

Effective Programming in Scala

How To Test Programs?

The approach that we showed in the previous lesson consisted in checking that for some input for which we know the correct answer, the program returns it.

```
test(”fibonacci”) {
  assertEquals(fibonacci(1), 0)
  assertEquals(fibonacci(2), 1)
  assertEquals(fibonacci(3), 1)
  assertEquals(fibonacci(4), 2)
  assertEquals(fibonacci(5), 3)
}
```

Is it enough to check 5 cases to conclude that the implementation is correct?

How To Test More Cases?

Writing each test example takes a bit of time. Generally, program domains are huge, it would be impossible to write test cases manually for the whole domain space.

You can make the effort of testing all the edge cases, but you would still miss a lot of cases.

Property-Based Testing

Another approach is to generate random input data.

How can we specify the expected result of calling a program with data that we don't know?

Property-Based Testing

Another approach is to generate random input data.

How can we specify the expected result of calling a program with data that we don't know?

We can only specify general **properties** that must be correct for all possible inputs (or a well delimited subset).

An example of property that we might want to check for the `fibonacci` function is that any fibonacci number is the sum of the two previous fibonacci numbers.

Getting Started With ScalaCheck

ScalaCheck is a library for doing property-based testing. It integrates with MUnit so that you only need to add the following dependency to your `build.sbt` file:

```
libraryDependencies += "org.scalameta" %% "munit-scalacheck" % "0.7.23" % Test
// Same as for unit testing
testFrameworks += new TestFramework("munit.Framework")
```

Defining a Test Suite

A test suite containing properties is a class that extends `munit.ScalaCheckSuite`:

```
// File src/test/scala/testing/ProgramProperties.scala
package testing

class ProgramProperties extends munit.ScalaCheckSuite:
  property("fibonacci(n) == fibonacci(n - 1) + fibonacci(n - 2)") {
    // TODO Write a property for the ‘fibonacci‘ method
  }
```

Writing a Property

```
package testing

import org.scalacheck.Prop.forAll

class ProgramProperties extends munit.ScalaCheckSuite:
  property("fibonacci(n) == fibonacci(n - 1) + fibonacci(n - 2)") {
    forAll { (n: Int) =>
      fibonacci(n) == fibonacci(n - 1) + fibonacci(n - 2)
    }
  }
```

The `forAll` method takes as parameter a function that receives an arbitrary value (here, `n`, of type `Int`) and returns a `Boolean` value.

Running the Tests From Sbt

Invoke the `test` task.

The library ScalaCheck generates some arbitrary input data and evaluates our properties.

Reading the Test Report (1)

```
sbt:testing> test
testing.ProgramProperties:
==> X testing.ProgramProperties.fibonacci  50.843s munit.FailException
13:    }
14:  }
15:

Failing seed: v8uYk_Kv8OXPcwPa_k9RlVuUzWFlgeu_6AM20O2QlRJ=
You can reproduce this failure by adding the following override to your suite:

  override val scalaCheckInitialSeed =
    "meIUZ8Riey9L_rSQW_BUK6JCG5XXNwpMnc8KxtP66mA="

Falsified after 6 passed tests.
> ARG_0: 1
```

Reading the Test Report (2)

The test report tells us that the property `fibonacci` failed after 6 passed tests.

The input value that falsified the property is `1`.

This means that:

```
fibonacci(1) != fibonacci(1 - 1) + fibonacci(1 - 2)
```

Which simplifies to:

```
fibonacci(1) != fibonacci(0) + fibonacci(-1)
```

Leveraging Property-Based Testing To Find the Edge Cases (1)

The problem is that there is no "zero" element or "minus one" element in the Fibonacci sequence. The first element has index 1, so indices below 1 are invalid.

We should exclude such input values from our tests.

(Alternatively, we could narrow the input type of the `fibonacci` method: it currently takes an `Int`, we could replace it with a type `PosInt` whose values would be guaranteed to be always greater than zero)

Explicit Input Data Generator

We can control how the input data is generated by explicitly supplying a generator:

```
import org.scalacheck.Gen
import org.scalacheck.Prop.forAll

class ProgramProperties extends munit.ScalaCheckSuite:

  val fibonacciDomain: Gen[Int] = Gen.choose(3, Int.MaxValue)

  property("fibonacci(n) == fibonacci(n - 1) + fibonacci(n - 2)") {
    forAll(fibonacciDomain) { (n: Int) =>
      fibonacci(n) == fibonacci(n - 1) + fibonacci(n - 2)
    }
  }
```

Leveraging Property-Based Testing To Find the Edge Cases (2)

Another property of Fibonacci numbers is that they are all positive numbers:

```
property("Fibonacci numbers are positive") {
  forAll(fibonacciDomain) { (n: Int) =>
    fibonacci(n) >= 0
  }
}
```

Leveraging Property-Based Testing To Find the Edge Cases (2)

Another property of Fibonacci numbers is that they are all positive numbers:

```
property("Fibonacci numbers are positive") {
  forAll(fibonacciDomain) { (n: Int) =>
    fibonacci(n) >= 0
  }
}
```

Unfortunately, running the tests reveals our `fibonacci` method sometimes returns negative numbers.

What happens here is that arithmetic on `Int` values (32-bit signed integers) is different from arithmetic on natural numbers.

In particular, adding two positive `Int` values can result in a negative value!

Finding the Domain of the `fibonacci` Method

In the case of the `fibonacci` sequence, the numbers almost double each time, meaning that the 32th Fibonacci number is close to 2^{32}, which can not be represented on an `Int` value.

In practice, we can use a worksheet to find the edge cases:

```
testing.fibonacci(32) // : Int = 1346269

testing.fibonacci(47) // : Int = 1836311903

testing.fibonacci(48) // : Int = -1323752223
```

Properties of the `fibonacci` Method

```
class ProgramProperties extends munit.ScalaCheckSuite:

  val fibonacciDomain = Gen.choose(1, 47)

  property("fibonacci(n) == fibonaci(n - 1) + fibonacci(n - 2)") {
    forAll(fibonacciDomain.suchThat(_ >= 3)) { (n: Int) =>
      fibonacci(n) == fibonacci(n - 1) + fibonacci(n - 2)
    }
  }

  property("Fibonacci numbers are positive") {
    forAll(fibonacciDomain) { (n: Int) =>
      fibonacci(n) >= 0
    }
  }
```

How To Find Properties?

Good properties may be hard to find or formulate.

You should look for **invariants** and **identities** (such as invertibility, idempotence, transformation relations, etc.).

You can learn more about this in the talk “Much Ado About Testing”, by Nicolas Rinaudo, 2019.

Summary

Property-based testing makes it easier to increase the coverage of the tested domain, and helps to find edge cases.

However, good properties can be hard to find (be careful to not re-implement the system under test!).