CMSC 330, Fall 2022

Property-Based Testing

Lecture Review
University of Maryland, College Park
Department of Computer Science

JC+CB

September 29, 2022

Contents

1 Introduction

Testing. It's important.

There are many ideas and patterns related to testing, many are beyond the scope of this lecture. Our focus is going to be on how we can *generalize* unit-testing. This begins with a quick discussion of what unit-testing is, then we will introduce property-based testing as a powerful generalization.

2 Unit Tests

Unit tests are powerful, luckily they are also relatively simple to understand! When working in `utop`, you may be doing unit testing already. Consider the following definition of a function to calculate the Fibonacci numbers:

```
let rec fib x =
  if x < 2
  then 1
  else (fib (x-1)) + (fib (x-2))
```

We may find ourselves running some sample invocations and seeing if the answer is what we expect. Perhaps we would test the following in `utop`:

```
utop # fib 5;;
- : int = 8
```

If we know what the result should be ahead of time, it might be worthwhile to check the value explicitly:

```
utop # fib 5 = 8;;
- : bool = true
```

Doing that on a few examples is good, and we definitely encourage it! The idea of Unit-Tests is that we can check these examples *programmatically*. So in a separate file we might have a bunch of these equality checks, each one being 'a unit test'.

2.1 OUnit

Unit testing frameworks (like `OUnit`, for OCaml) provide functionality around this simple idea. The core idea is unchanged from above, but the framework allows for naming of specific tests and 'nicer' error reporting. Consider this small example (note that `open` is how we import libraries, in this case a library for testing):

```
open OUnit2

let rec sum xs =
  match xs with
  | []    -> 0
  | y::ys -> y + sum ys

let tests = "test suite for sum" >::: [
  "empty" >:: (fun _ -> assert_equal 0 (sum []));
  "singleton" >:: (fun _ -> assert_equal 1 (sum [1]));
  "two_elements" >:: (fun _ -> assert_equal 3 (sum [1; 2]));
]
```

This allows us to programmatically run these tests, and when we change our implementation of `sum`, we can run these tests to make sure we haven't broken our code. Unit tests are very powerful and consistently gaining wider adoption in industry. In fact, unit testing is considered a 'best practice' in industrial software engineering today. We aren't going to make you learn `OUnit` in this course, but if you're interested, you can learn more at `https://ocaml.org/p/ounit2/2.2.3/doc/index.html`.

The drawback of Unit Testing

While powerful, unit testing does suffer from a very particular limitation: *you have to think of the things to check*. Thinking up examples is difficult, and on top of that, there's always the chance that your test is wrong. Because of this, programmers often write unit tests for the cases that are easy to think about. As an unfortunate consequence, the edge cases that are hard to think about are less likely to be written as tests!

Finding the edge-cases

There are a few techniques to improve a test-suite's ability to take edge cases into account. One of the more popular technologies is known as *code coverage*[1]. In short, code coverage is about measuring how much of your code actually gets run/executed when you test it. The idea is that the more of your code that gets run, the more of the possible edge cases you've taken into consideration. Different languages and testing frameworks provide different ways of measuring code coverage. While code coverage is a useful *metric*, it does not solve the problem, you still have to think of the examples!

Property-based testing seeks to solve the problem of test-case generation by *generalizing* unit tests.

3 Property-Based Testing

Unit tests work by testing code on particular input *examples*. These examples must be generated by *something*, usually the programmer who wrote the test, or some input that caused a crash (a hard-earned example!). As mentioned above, thinking of examples is difficult, especially for non-obvious edge-cases.

Property-based testing aims to free the programmer from the task of example generation. It achieves this with the following trade: If you can think of a *property* that your code should satisfy, the property-based testing framework will generate as many examples as you'd like. So our problem shifts from 'thinking of examples' to 'thinking of properties'.

3.1 Properties

Much of the code we right is written with certain assumptions in mind. Often these assumptions can be expressed as *properties*[2]. A 'property' is simply 'a thing we expect to be true' about our code, usually as it relates to some other code or data structure. Let's look at some examples of properties:

- reversing a list should not change the number of elements in this list

[1] The wikipedia article is pretty good for describing code coverage: `https://en.wikipedia.org/wiki/Code_coverage`.

[2] And sometimes they can't!

- a sorted list should have its minimum element as the first element
- combining two hash tables should not result in a hash table that is larger than the two independent sizes added together
- adding an element to a set, and then looking up that element, should succeed

Take a moment and try to think of some properties of code you've written in the past.

4 QCheck

Because OCaml is a functional language, it is unlikely to surprise you that we are going to write our properties as functions. Consider the following property: Taking the `floor` (rounding a `float` down to an `int`) of a number should result in a number that is less than or equal to the original number. Let's write that as a function:

```
(fun f -> floor f <= f)
```

That's it, that's the property.

Now we want to have our computer *generate* examples for this property. Each individual example can be seen as a single unit test, so by thinking in terms of properties we can have the computer generate *an arbitrary number of unit tests*. There is an OCaml library call `QCheck` that will generate examples for a given property. It looks like this:

```
let round_down_test = Test.make float (fun f -> floor f <= f);;
```

Here we're using the `Test.make` function, which is provided by the `QCheck` library, to generate a bunch of `float`s to test this property on. We would then use `QCheck2_runner.run_tests` to actually run the tests. Putting it all together in `utop` looks like the following:

```
utop # #require "qcheck";;
utop # open QCheck;;
utop # let round_down_test = Test.make float (fun f -> floor f <= f);;
val round_down_test : Test.t = QCheck2.Test.Test <abstr>
utop # QCheck_runner.run_tests [round_down_test];;
==============================================================================
success (ran 1 tests)
- : int = 0
```

Let's put a bug in our implementation to see if this can find it:

```
utop # let round_down_test = Test.make float (fun f -> floor (f +. 0.1) <= f);;
val round_down_test : Test.t = QCheck2.Test.Test <abstr>
utop # QCheck_runner.run_tests [round_down_test];;
random seed: 414488010

--- Failure ------------------------------------------------------------------

Test anon_test_1 failed (0 shrink steps):

-3.74649808603e-07
================================================================================
failure (1 tests failed, 0 tests errored, ran 1 tests)
- : int = 1
```

This is showing us that on the input of `-3.74649808603e-07`, our property did not hold.

4.1 Other Properties

Earlier we had a list of potential properties, take each one and write it as a property in OCaml. For example, here's the first one "reversing a list should not change the number of elements in this list". As a property it would look like the following:

```
(fun xs -> length xs = length (reverse xs))
```

Having `QCheck` generate lists for us is very similar to generating `float`s, here's the `utop` session:

```
utop # let len_prop = (fun xs -> List.length xs = List.length (rev xs));;
val len_prop : 'a list -> bool = <fun>
utop # let len_test = Test.make (list int) len_prop;;
val len_test : Test.t = QCheck2.Test.Test <abstr>
utop # QCheck_runner.run_tests [len_test];;
================================================================================
success (ran 1 tests)
- : int = 0
```

Looks like OCaml's implementation of 'rev' does the right thing (at least with regard to this property).

Exercise: Write an implementation of 'reverse' that would not satisfy this property (perhaps it drops an element), and see what `QCheck` reports.

Exercise: Write the other properties and test them.

5 Limitations of Property-Based Testing

Like nearly all technical solutions, property-based testing is not a panacea. Not all code can be described via properties. Stateful code (i.e. code that uses mutable variables) and code that executes side-effects (e.g. reading/writing from/to files) is particularly difficult to describe as properties.

The reality is that property-based testing and unit testing both have their place. Where code can be described via properties, property-based testing provides an arbitrary number of unit tests 'for free'. For code where property-based testing is not realistic, unit tests are the right approach.