Chapter 1

Property Based Testing

Testing, 1, 2, 3, Testing 1,2,3.

Micheal Czech

1.1 Preface

I will first say that you should totally read José Calderon's wonderful notes from when we taught this course together in Fall 2022. They talk about property based testing in OCaml. Thus, this note set is more clarifications on those notes rather than a standalone thing.

1.2 Introduction

Testing is important. We say this all the time let us be very explicit as to why it's important.

- Testing can prevent you from getting fired.
- Testing can help you become a better project manager.
- Testing helps prevent your company (or you if you're an entrepreneur and trying to make a startup) from losing (potentially) millions of dollars.
- Testing helps you maintain integrity and can help make sure you don't get hacked (and lose money).
- Testing as you go can help prevent integration hell
- If nothing else, as a student, your grade is sometimes dependent on testing

Ultimately testing can help up save money and write better, more secure code. The goal of learning about Property Based Testing (PBT) is to give you another tool in your toolbox to help you test more thoroughly (and perhaps even more efficiently).

1.3 The problem

Testing is important yet having been a student, a TA, engineer, and a teacher, I know most people have a hard time testing. The reason people have a hard time and the problem we are trying to solve is two-fold: testing is difficult, and people do not like to test (probably because it is difficult).

Why is it so difficult? I believe it has to do with how we are initially taught to test. We are very used to and familiar with writing unit tests. A unit test is where we come up with an input (say 2) and then we have to calculate what the output should be (suppose negation leads to -2). We don't want to do this. We write software to calculate an answer so we don't

have to. Additionally, it can be really hard to think up edge or corner cases. Thus, we introduce the idea of PBT as a way to generalize unit testing.

In particular, we will use PBT to help in the following manner:

- speed up testing
- help catch weird edge cases that are hard to come up with

1.4 Property Based Testing

To property test, we need to shift our mindset from units to properties. A unit is often defined as an individual entity. So instead of thinking about individual inputs, we ought to think about groups (sets!) of inputs. By thinking of sets of inputs, we can no longer say something about the individual, but can only talk about the group as a whole.

Suppose we want to test our `square` function: a function that takes in an integer and returns the mathematical square of that integer.

```
1 int square(int i){
2   return i * i;
3 }
```

If we are going to think of sets of inputs, we can start thinking about what **relations** exist between the input and output of the function (the domain and codomain). For example: we know that an even number squared should be even, and odd number squared should be odd. That right there is a property. To implement testing on this property we need a few things:

- A property (what property are we testing)
- A relation function (a function that describes the **relation** between domain and codomain)
- A generator (a way to create the input set- the domain)

1.4.1 The Property

We already have the property (the output's parity should match the input's parity).

1.4.2 The Relation

We now need the relation function and generator. The relation function should be an *encoding* of the property we are trying to test. For example, if we were going to do a literal translation of the property, we would probably get something like the following:

```
1 int relation(int i){ // i in an element of the domain
2   return square(i)%2 == i%2;
3 }
```

Notice that we return a boolean (int in C). If the property holds for the input, the function returns true and false otherwise. This is because we are asking, "does the property hold for input `i`, yes or no"?

1.4.3 The generator

We then need to make the generator. The generator will need to make the domain. Typically, the generator will generate a **finite** set of **random** input.

```
1 srand(time(NULL));
2 int* generator(int i){ // i is cardinality of domain
3   int* ret = malloc(sizeof(int)*i);
4   for (int j =0; j < i; j++){
```

```
    ret[j] = rand(); //% 46340 so we don't get int overflow when calling square
  }
  return ret;
}
```

1.4.4 Putting it All Together

Once we have all our parts, we can put it together to build a pbt.

```
1 void student_test(){
2   int num_test_cases = 100;
3   int* domain = generator(num_test_cases);
4   for (int i = 0; i < num_test_cases; i++){
5     assert(relation(domain[i]));
6   }
7 }
```

All done. We now have "written" 100 test cases. We can make a hundred more by changing line 2. It is important to note that this is not the only way to test this property. We could have done by testing each parity separately:

```
int even_relation(int i){
  return square(i)%2 == 0;
}
int odd_relation(int i){
  return square(i)%2 == 1;
}
```

This also means we would need to modify our generator:

```
srand(time(NULL));
int* even_generator(int i){
  int* ret = malloc(sizeof(int)*i);
  for (int j =0; j < i; j++){
    ret[j] = rand()*2;
  }
  return ret;
}

int* odd_generator(int i){
  int* ret = malloc(sizeof(int)*i);
  for (int j =0; j < i; j++){
    ret[j] = rand()*2 + 1;
  }
  return ret;
}
```

1.5 PBT Libraries

PBT is pretty useful and so of course there exist libraries that help with PBT. They are all mostly derivatives from Haskell's QuickCheck library since PBT originated in Haskell.

- Python: Hypothesis
- Ocaml: qcheck
- Rust: proptest

1.5.1 Aside: Type systems and PBT

Properties that we want to test are typically language agnostic. But with languages that have a more "loose" type system like Python, we can create properties based on types.

```
def relation(i):
  return type(i) == type(square(i)) == int
}
```

1.5.2 PBT in Python

First, we need to import some functions from the `hypothesis` module.

```
from hypothesis import given, strategies as st, example
}
```

What we are importing here are three things:

- `given`: this is a flag for the test which will tell `hypothesis` that the test should be treated as a property based test
- `strategies`: this specifies the type of data to generate
- `examples`: this is a flag that will test a particular example (can also be used as documentation to show example inputs)

For example, if we wanted to test the `square` function in python, we would do the following:

```
@given(st.integers())       # this is a hypothesis test, and integers should be generated
@example(0)                 # it will always test with 0
def test_square_parity(s):
  assert(s%2 == square(s)%2)
}
```

If we had multiple inputs we wanted to test we could do the following:

```
@given(st.integers(), st.integers()) # will be in the same order as the arguments to the
    function
@example(2,2)
def test_ints_are_commutative(x, y):
    assert x + y == y + x
```

1.5.3 PBT in OCaml

When making PBT, the dune build system will include qcheck as as a required library.

```
(libraries qcheck) # will be in a dune file
```

However, if you wanted to use it in `utop`, you will have to do the following:

```
#require "qcheck";;
open QCheck;;
```

When you are ready to start testing, we can start making our PBT.

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

In this example, we are telling `qcheck` to make a `float` type.

1.5.4 PBT in Rust

When making PBT, the rust crate system will include proptest as a required library.

```
// will be in Cargo.toml
[dev-dependencies]
proptest = "1.0.0"
```

Once that is done, then we need to include the proptest macros to our test file:

```
use proptest::prelude::*;

proptest! {
    #[test]
    fn test_square(x in 0u32..10000){ // generate a u32 in range from 0 to 10000
       assert!(sqaure(x) >= 0);
    }
}
```

1.6 When things go wrong

It is important to note that if any of these 3 things (property, relation function, generator) is incorrect, then we may get wrong results (false positives or false negatives). So PBT is not infallible, nor is it meant to be a full replacement of unit testing.

For example, we can still come up with things that sound like properties but are not true: "Reversing a list will not be the same as the input list". This is not a valid property because both the empty list and the list of size 1 would result in the initial list if reversed.

We can also encode or write the relation function incorrectly.

```
int relation(int i){
  return square(i)%2 == 0; // should be parity of i
}
```

The generator could be wrong, or we use the wrong one. We could use the even generator on the initial parity property.

```
srand(time(NULL));
int* even_generator(int i){
  int* ret = malloc(sizeof(int)*i);
  for (int j =0; j < i; j++){
    ret[j] = rand()*2;
  }
  return ret;
}

int relation(int i){
  return square(i) % 2 == i % 2;
}
```

In this case, we don't test odd input and so we could be missing a bug here.

It is also just possible that our property introduces new edge cases that we cannot cover with a property. There are also certain things we cannot (or it's very very hard) to encode as a property (things like user input).

This is why we have to think of **multiple properties** much like how we have to think of multiple units.r