Property Based Generator Testing

Justice Martinez CS-370

December 16, 2017

1 Introduction

Working with functional languages can be tedious and hard, if a function can only take in a certain type of parameter but needs to be true for any parameter of that type. Functional languages can also be easier to work with in the sense that reasoning what the code is doing can be easy, but not always. Having a complicated in a functional language can still be complicated to reason through and being able to test these statements, and prove them to be right, is key to making a program run properly. The problem with testing is that there are so many possibilities of inputs that it is a large task to take this on single-handedly and running each test by hand. For this paper Haskell will be used along side the QuickCheck library for testing Haskell functions. This paper will define and cover property based testing, give a brief overview of the QuickCheck library, and discuss the implications of a new language called Luck that will improve user interaction with the QuickCheck library.

1.1 Property Based Generator Testing

In functional languages there are only a few ways to test properties for correctness; one being random based property testing where type of input that the property should be proven for is given by the user and inputs of these types are chosen randomly [1, 2, 3], the second being search based property testing where again the type of input that the property should be proven for is given by the user and a search priority is given for the testing to be run on [3], and lastly targeted testing which is a combination of the two types of testing that is being researched but is not fully developed yet [3].

For this paper only random based property testing will be looked at. Random based property testing (RBT) is used to test whether a statement hold true for all randomly generated inputs that match the given criteria. This type of testing allows the user to specify what should be tested for and allows an easy quick way to run a variety of tests without having to generate each individual test case by hand. As properties get more complex however there is a need for better generators to accurately show validity. A lot of problems with random generator testing is that conditional statements require additional criteria before the test is even run, which can skew the results if too many inputs are thrown out due to not meeting the criteria.

```
prop_reverse :: [Int] -> Bool
prop_reverse xs = reverse (reverse xs) == xs
```

Figure 1: Example of QuickCheck property for a reversed reversed list equaling the original list.

This is where generators come into play. QuickCheck has built in generators that do a good job of generating simple inputs, such as those with no conditional constraints. To get around conditional tests not being representative by these built in generators, user must create their own that can be used to test their already coded predicate. To be able to do this effectively the user must have a decent understanding of the lambda calculus that is derived from their problem to be able to make a generator that correctly distributes test cases for the range of possibilities.

1.2 QuickCheck

A small example of QuickCheck, is testing if the reverse of a reversed list is the original list as shown in Figure 1 [4]. This example shows that the required inputs should be a list of Int types, and the QuickCheck is testing the property that says if the given list is reversed and then that list is reversed that should equal the original list. QuickCheck will generate 100 random lists of Int types, and will test the property starting with the simplest list. This property holds true and it is easy to reason through this. Another simple example that proves to be false can be shown in Figure 2 [4], simply by switching which reversed list is first on the right side of the equation makes this statement true, but as is QuickCheck will usually find a counterexample to this property within the first five tests run.

As stated in the previous section once the predicates become more complex, the built in generators in the QuickCheck library do not create a random sample of inputs that represent all possible inputs. For conditional statements ($p \to q$) test cases that do not meet the p statement will be thrown out, but QuickCheck still only generates 100 random inputs. So if the conditional statement is constrictive then QuickCheck may only run a hand full of tests before giving up, and while the predicate may pass for those inputs there may be other inputs that were not attempted. One way to do this is to code generators that will fit the pre-conditional statement. This can be very difficult depending on what is being tested. With QuickCheck writing generators is inconvenient since you have to create the object you are testing as well as the test. This involves writing two separate parts of code that can be very abstract in their implementation.

Given the length of time that QuickCheck has been available there are surprisingly little updates that have been made to simplify the coding experience. It still remains fairly complicated to write and is just as complicated to read.

```
prop_revapp :: [Int] -> [Int] -> Bool
prop_revapp xs ys = reverse (xs++ys) == reverse xs ++ reverse ys
```

Figure 2: Example of QuickCheck property for a reversed concatenation of lists.

2 Luck

Recently there has been the development of a new language that is to be used with the QuickCheck library. This new language, called Luck, is aimed to create a more user friendly process in creating predicates to test and the generators needed to test the predicates. Luck is just in early developmental stages so currently there is no code yet to see if their syntax will indeed create a simpler process, but they aim to merge the two main processes needed for random based generator testing and making them one seem less process. Along with combining these two processes the developers of Luck are trying to make these predicates easier to read and understand what is happening in the code. In their most recent article the developers of Luck have laid out basic semantics for their language and proven that these semantics will work with small examples [2].

3 Conclusions

Having worked with QuickCheck very briefly it has been easy to see how much of learning curve there is too learning how to use it effectively. Even getting used to working with simple examples in QuickCheck was a challenge. The concept for Luck seems like a really interesting idea. Working with functional languages has been interesting and something that I like to use since it is so closely related to math. Finding out about this type of testing was really intriguing but working with it and learning it seemed to be impossible, so the reasoning for a language such as Luck is very greatly needed. I hope that this concept gets refined in implemented very quickly.

References

[1] Zilin Chen Liam OConnor Gabriele and Keller Gerwin Klein Gernot Heiser. The cogent case for property-based testing. 2017.

[2] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 114–129. ACM, 2017.

[3] Andreas Löscher and Konstantinos Sagonas. Targeted property-based testing. 2017.

[4] Pedro Vasconcelos. An introduction to quickcheck testing. `www.schoolofhaskell.com/user/pbv/an-introduction-to-quickcheck-testing/`, 2015.