test.check

Justin Overfelt

Introduction

Test.check is a tool for performing property-based testing (PBT). PBT is an approach to testing that relies on creating universal quantifications, or properties, that hold true for all input to a particular function. In order to test a function in this manner, a PBT framework will typically be used to generate data, define a property or set of properties to test, and test that the property holds over a user-defined number of random samples from the generated data. Specifically, test.check is an implementation of a PBT framework for the Clojure language. Clojure is a member of the Lisp family of languages. However, it is different from most other Lisps in that it runs on the Java Virtual Machine (JVM), and so users are able to import and call libraries written in any other language that also runs on the JVM. Test.check includes several functions and macros that allow for the definition of properties and the generation of inputs, including complex inputs such as user-defined types and collections. Test.check also provides generator combinators, which allow the tester to take a generator and create new generators by filtering return values using a predicate (such-that) or by transforming each generated value using a function (fmap) among other things. The rich generator and generator combinator support make generating complex structures very straightforward. Examples of generator usage can be found in the Appendix.

It is important to differentiate this approach from fuzzing (random testing). Property based testing assumes some knowledge about the code being tested, and thus can be used as a white box test. Also, many property based testing tools use a technique called shrinking in order to find the smallest possible failure when a property does not hold. Shrinking is generally not a technique employed by fuzzing frameworks, and thus determining why an input caused a failure can be daunting if the input is large.

In order to explore the functionality of test.check, several Clojure libraries were placed under test. These libraries were selected specifically to demonstrate areas for which test.check is particularly well suited. In a conference presentation, the creator of test.check (Reid Draper) mentions three types of test cases where he believes test.check shines.

1. “round trip”: Ensuring that applying a function to some data and then applying the inverse of that function produces the original data
2. “trusted implementation”: Testing that an optimized or otherwise modified version of a standard algorithm produces results which match those of a standard library implementation.

3. “input/output relation”: Testing that some property holds which relates the inputs of a function to its outputs.

To that end, the data.json library and the data.fressian library were put under test as examples of the “round trip” scenario. Data.json is a Clojure library that provides functions for encoding and decoding data in JavaScript Object Notation; it converts JSON strings to Clojure data structures and vise versa. Data.fressian is a library which allows for the encoding and decoding for a binary transfer format called Fressian. Encoding something using these libraries and then decoding it should produce the original value. For the “trusted implementation” scenario, the data.avl library has been selected. Data.avl provides implementations of sorted maps and sets that are supposed to be identical to those provided in Clojure’s standard library, with the exception that the sorted data structures in data.avl are backed by AVL trees. These data structures should perform exactly the same as those in the standard library. Finally, an example of the “input/output relation” can be found in the data.priority-map library. This library provides maps (associative arrays) that function exactly like those in the standard library, with the exception that they are sorted by value. Therefore, a sequence of values from a map generated from this library should be sorted.

Observations

Test.check is installed like any other Clojure library. If one is using the de facto Clojure build tool, Leiningen, installing it is simply a matter of including the dependency information in Leiningen’s project configuration file (project.clj). Again, this code is contained in the Appendix.

In using test.check to test properties on the library code mentioned above, it became apparent that a different kind of thinking was required for successful use. It is very easy to catch oneself duplicating the implementation of code while testing that code. This is a mistake. Instead, the tool teaches the tester to think in terms of higher level properties. It is important to note, however, that not every function is amenable to property based testing. PBT is best employed against pure (stateless) functions, as creating universal quantifications over functions which depend on outside state is difficult. The advantage is that concentrating state in a small part of one’s program and keeping most functions pure is good programming practice anyhow.

During testing, test.check was found to have several advantages over traditional unit testing. First and foremost, the number of tests run is configurable. In traditional unit testing, adding more tests takes effort that is linear in the number of tests added. Adding more tests to a particular property is trivial - just increase the user-configured number of tests. This opens up all sorts of possibilities, like running a smaller number of tests locally or in a continuous integration system while having a special job that runs a large number of tests during off hours. Any reported failure can easily be converted into a unit test and run with the regular test suite. This highlights another

advantage of property based testing: it can be integrated into an already existing test suite without modification to existing tests or existing code. Other tools exist for checking that properties hold across all runs of code, but they frequently require modification of source code and/or the introduction of a precompile step into the build pipeline. Both of these can be difficult on the large, legacy codebases frequently seen in industry.

Aside from the fact that PBT is not easily applicable to all types of functions, the other main issue with test.check is really an issue with Clojure itself: poor error messages. Because the Clojure language is shipped as a Java library, stack traces and errors must make their way up through Java, into the Clojure runtime, and finally through library code and any user code. This makes finding the location of errors difficult. Poor error messages could be improved, but it is likely that would have to happen at a language-wide level.

Finally, while a Clojure library was studied, PBT frameworks exist for many modern languages, including Java and JavaScript. In fact, most PBT frameworks were inspired by a Haskell library called QuickCheck.

Conclusions

Test.check is a valuable tool to gain greater confidence in a particular piece of code. It is not, however, a silver bullet and is at its best when combined with other testing methodologies. The libraries under test performed well - no property violations were observed. It seems likely that property based testing will become more popular as its benefits are more widely observed.

Works Cited

https://github.com/clojure/test.check/tree/master/doc
https://www.youtube.com/watch?v=JMhNINPo__g

Appendix

Code, output and installation instructions are submitted alongside this report