Assertion-based testing with Quickcheck

Introduction

Quickcheck was originally a package for the language Haskell aiming to simplify the writing of tests. The main idea is the automatic generation of tests based on assertions a function needs to satisfy and the signature of that function. The idea spread to other languages and is now implemented in R with this package. Because of the differences in type systems between Haskell and other languages, the original idea morphed into something different for each language it was translated into. In R, the main ideas retained are that tests are based on assertions and that the developer should not have to specify the inputs and output values of a test. The difference from Haskell is that the user needs to specify the type of each variable in an assertion with the optional possibility to fully specify its distribution. The main function in the package, `test`, will randomly generate input values, execute the assertion and collect results. The advantages are multiple:

- each test can be run multiple times on different data points, improving coverage and the ability to detect bugs, at no additional cost for the developer;
- tests can run on large size inputs, possible but impractical in non-randomized testing;
- assertions are more self-documenting than specific examples of the I/O relation – in fact, enough assertions can constitute a specification for the function being tested, but that's not necessary for testing to be useful;
- it is less likely for the developer to use implicit assumptions in the selection of testing data – randomized testing "keeps you honest".

First example

Let's start with something very simple. Let's say we just wrote the function `identity`. Using the widely used testing package `testthat`, one would like to write a test like:

```
library(testthat)
```

```
test_that("identity test", expect_identical(identity(x), x))
```

That in general doesn't work because `x` is not defined. What was meant was something like a quantifier *for all legal values of* x, but there isn't any easy way of implementing that. So a developer has to enter some values for `x`.

```
for(x in c(1L, 1, list(1), NULL, factor(1)))
  test_that("identity test", expect_identical(identity(x),x))
rm(x)
```

But there is no good reason to pick those specific examples, testing on more data points or larger values would increase the clutter factor, a developer may inadvertently inject unwritten assumptions in the choice of data points etc. `quickcheck` can solve or at least alleviate all those problems:

```
library(quickcheck)
test(function(x = rinteger()) identical(identity(x), x))
```

```
Pass  function (x = rinteger())
 identical(identity(x), x)
```

```
[1] TRUE
```

We have supplied an assertion, that is a function with defaults for each argument, at least some set using random data generators, and returning a length-one logical vector, where `TRUE` means *passed* and `FALSE` means *failed*. What this means is that we have tested `identity` for this assertion on random integer vectors. We don't have to write them down one by one and later we will see how we can affect the distribution of such vectors, to make them say large in size or value, or more likely to hit corner cases. We can also repeat the test multiple times on different values with the least amount of effort, in fact, we have already executed this test 10 times, which is the default. But if 100 times is required, no problem:

```
test(function(x = rinteger()) identical(identity(x), x), sample.size = 100)
```

```
Pass  function (x = rinteger())
 identical(identity(x), x)
```

```
[1] TRUE
```

Done! You see, if you had to write down those 100 integer vectors one by one, you'd never have time to. But, you may object, `identity` is not supposed to work only on integer vectors, why did we test only on those? That was just a starter indeed. Quickcheck contains a whole repertoire of random data generators, including `rinteger`, `rdouble`, `rcharacter` etc. for most atomic types, and some also for non-atomic types such as `rlist` and `rdata.frame`. The library is easy to extend with your own generators and offers a number of constructors for data generators such as `constant` and `mixture`. In particular, there is a generator `rany` that creates a mixture of all R types (in practice, the ones that `quickcheck` currently knows how to generate, but the intent is all of them). That is exactly what we need for our identity test.

```
test(function(x = rany()) identical(identity(x), x), sample.size = 100)
```

```
Pass  function (x = rany())
 identical(identity(x), x)
```

```
[1] TRUE
```

Now we have more confidence that `identity` works for all types of R objects.

Defining assertions

Unlike `testhat` where you need to construct specially defined *expectations*, `quickcheck` accepts run of the mill logical-valued functions, with a length-one return value. For example `function(x) all(x + 0 == x)` or `function(x) identical(x, rev(rev(x)))` are valid assertions – independent of their success or failure. If an assertion returns TRUE, it is considered a success. If an assertion returns FALSE or generates an error, it is considered a failure. For instance, `function(x = rcharacter()) stop()` is a valid assertion but always fails. How do I express the fact that this is its correct behavior? `testthat` has a rich set of expectations to capture that and other requirements, such as printing something or generating a warning. Derived from those, `quickcheck` has a rich set of predefined assertions, returned by the function `expect`:

```
test(
  function(x = rcharacter()) expect("error", stop(x)))
```

```
Pass  function (x = rcharacter())
 expect("error", stop(x))
```

```
[1] TRUE
```

By executing this test successfully we have built confidence that the function `stop` will generate an error whenever called with any `character` argument. There are predefined `quickcheck` assertions defined for each `testthat` expectation, with a name equal to the `testthat` expectation, without the "expect_" prefix. We don't see why you'd ever want to use `expect("equal", ...)`, but we threw it in for completeness.

What to do when tests fail

`quickcheck` doesn't fix bugs for you, but tries to get you started in a couple of ways. The first is its output:

```
test(function(x = rdouble()) mean(x) > 0, stop = TRUE)
```

```
FAIL: assertion:
function (x = rdouble())
mean(x) > 0FAIL: assertion:
function (x = rdouble())
mean(x) > 0FAIL: assertion:
function (x = rdouble())
mean(x) > 0FAIL: assertion:
function (x = rdouble())
mean(x) > 0FAIL: assertion:
function (x = rdouble())
mean(x) > 0FAIL: assertion:
function (x = rdouble())
mean(x) > 0

Error in test(function(x = rdouble()) mean(x) > 0, stop = TRUE): load("/home/craig/Documents/Books/R Te
```

Its output shows that about half of the default 10 runs have failed and then invites us to load some debugging data. Another way to get at that data is to run the test with the option `stop = FALSE` which doesn't produce an error. This is convenient for interactive sessions, but less so when running `R CMD check`. In fact, the default for the `stop` argument is `FALSE` for interactive sessions and `TRUE` otherwise, which should work for most people.

```
test.out = test(function(x = rdouble()) mean(x) > 0, stop = FALSE)
```

```
FAIL: assertion:
function (x = rdouble())
mean(x) > 0FAIL: assertion:
function (x = rdouble())
mean(x) > 0FAIL: assertion:
function (x = rdouble())
mean(x) > 0FAIL: assertion:
function (x = rdouble())
mean(x) > 0FAIL: assertion:
function (x = rdouble())
mean(x) > 0FAIL: assertion:
function (x = rdouble())
mean(x) > 0
```

```
test.out
```

```
$assertion
function (x = rdouble())
mean(x) > 0

$env
NULL

$cases
$cases[[1]]
NULL

$cases[[2]]
$cases[[2]]$x
[1] -0.4115108  0.2522234 -0.8919211  0.4356833 -1.2375384 -0.2242679
[7]  0.3773956  0.1333364  0.8041895


$cases[[3]]
$cases[[3]]$x
[1]  0.50360797  1.08576936 -0.69095384 -1.28459935  0.04672617 -0.23570656
[7] -0.54288826 -0.43331032 -0.64947165


$cases[[4]]
$cases[[4]]$x
 [1]  1.1519118  0.9921604 -0.4295131  1.2383041 -0.2793463  1.7579031
 [7]  0.5607461 -0.4527840 -0.8320433 -1.1665705 -1.0655906 -1.5637821


$cases[[5]]
$cases[[5]]$x
 [1]  0.83204713 -0.22732869  0.26613736 -0.37670272  2.44136463
 [6] -0.79533912 -0.05487747  0.25014132  0.61824329 -0.17262350
[11] -2.22390027 -1.26361438  0.35872890


$cases[[6]]
$cases[[6]]$x
[1] -0.94064916 -0.11582532 -0.81496871  0.24226348 -1.42509839  0.36594112
[7]  0.24841265  0.06528818  0.01915639


$cases[[7]]
$cases[[7]]$x
 [1] -0.6490101 -0.1191688  0.6641357  1.1009691  0.1437715 -0.1177536
 [7] -0.9120684 -1.4375862 -0.7970895  1.2540831


$cases[[8]]
NULL

$cases[[9]]
```

```
NULL

$cases[[10]]
NULL
```

The output is a list with three elements:

- the assertion that failed
- a list of in-scope variables that could have affected the result – this is work in progress and shouldn't be trusted at this time
- a list of arguments passed to the assertion

My recommendation is to write assertions that depend exclusively on their arguments and are deterministic functions, and leave all the randomness to `quickcheck` and its generators. This is because the first step in fixing a bug is almost always to reproduce it, and non-deterministic bugs are more difficult to reproduce. The `test` function seeds the random number generator so that every time it is called it will rerun the same tests, that is call the assertion with the same arguments, run after run. So I guess we should call it pseudo-random testing to be precise. Let's go in more detail on the `cases` element. It is a list with an element for each run, which has a value of `NULL`, if the run was successful, and a list of arguments passed to the assertion otherwise. In this case runs 2 through 7 failed. We can replicate it as follows.

```
repro(test.out)
```

```
debugging in: (function (x = rdouble())
mean(x) > 0)(x = c(-0.411510832795067, 0.252223448156132, -0.891921127284569,
0.435683299355719, -1.23753842192996, -0.224267885278309, 0.377395645981701,
0.133336360814841, 0.804189509744908))
debug: mean(x) > 0
exiting from: (function (x = rdouble())
mean(x) > 0)(x = c(-0.411510832795067, 0.252223448156132, -0.891921127284569,
0.435683299355719, -1.23753842192996, -0.224267885278309, 0.377395645981701,
0.133336360814841, 0.804189509744908))

[1] FALSE
```

This opens the debugger at the beginning of a failed call to the assertion. Now it is up to the developer.

Modifying or defining random data generators

There are built in random data generators for most built-in data types. They follow a simple naming conventions, "r" followed by the class name. For instance `rinteger` generates a random integer vector. Another characteristic of random data generators as defined in this package is that they have defaults for every argument, that is they can be called without arguments. Finally, the return value of different calls are statistically independent. For example""

```
rdouble()
```

```
 [1] -0.5080551 -0.2073807 -0.3928079 -0.3199929 -0.2791133  0.4941883
 [7] -0.1773305 -0.5059575  1.3430388 -0.2145794 -0.1795565 -0.1001907
[13]  0.7126663 -0.0735644
```

```
rdouble()
```

```
[1] -0.68166048 -0.32427027  0.06016044 -0.58889449  0.53149619 -1.51839408
[7]  0.30655786 -1.53644982 -0.30097613
```

As you can see, both elements and length change from one call to the next and in fact they are both random and independent. This is generally true for all generators, with the exception of the trivial generators created with `constant`. Most generators take two arguments, `element` and `size` which are meant to specify the distribution of the elements and size of the returned data structures and whose exact interpretation depends on the specific generator. In general, if the argument `element` is a value it is construed as a desired expectation of the elements of the return value, if it is a function, it is called with a single argument to generate the elements of the random data structure. For example

```
rdouble()
```

```
[1] -0.65209478 -0.05689678 -1.91435943  1.17658331 -1.66497244 -0.46353040
[7] -1.11592011 -0.75081900
```

generates some random double vector. The next expression does the same but with an expectation equal to 100

```
rdouble(element = 100)
```

```
 [1] 100.01740  98.71370  98.35939 100.45019  99.98144  99.68193  99.07064
 [8]  98.51254  98.92481 101.00003  99.37873  98.61557 101.86929 100.42510
[15]  99.76135 101.05848
```

and finally this extracts the elements from a uniform distribution with all parameters at default values.

```
rdouble(runif)
```

```
 [1] 0.26787813 0.76215153 0.98631159 0.29360555 0.39935111 0.81213152
 [7] 0.07715167 0.36369681 0.44259247 0.15671413 0.58220527 0.97016218
```

The same is true for argument `size`. If not a function, it is construed as a length expectation, otherwise it is called with a single argument equal to 1 to generate a random length.

First form:

```
rdouble(size = 1)
```

```
[1] -0.9289711 -0.2941965 -0.6149503 -0.9470758
```

```
rdouble(size = 1)
```

```
[1] -0.03472603
```

Second form:

```
rdouble(size = function() 10 * runif(1))
```

```
[1] -0.20618900 -0.57429541 -1.39016604 -0.07041738 -0.43087953 -0.59222537
[7]  0.98111616
```

```
rdouble(size = function() 10 * runif(1))
```

```
[1]  0.5210227 -0.1587546  1.4645873 -0.7660820 -0.4302118 -0.9261095
[7] -0.1771040
```

A shorthand for the above expression is:

```
rdouble(size = ~10*runif(1))
```

```
[1]  2.02476139 -0.70369425  0.96079238  1.79048505 -1.06416516  0.01763655
```

Two dimensional data structures have the argument `size` replaced by `nrow` and `ncol`. Nested data structures have an argument `height`. All of these are intended to be expectations as opposed to deterministic values but can be replaced by a generator, which gives you total control. If you need to define a test with a random vector of a specific length as input, use the generator constructor `constant`:

```
rdouble(size = constant(3))
```

```
[1] -0.3899086 -0.4908328 -1.0457177
```

```
rdouble(size = constant(3))
```

```
[1] -0.8962113  1.2693872  0.5938409
```

Or, since "conciseness is power":

```
rdouble(size = ~3)
```

```
[1]  0.7756343  1.5573704 -0.3654018
```

Without the `~` it would be an expected size, with it it is deterministic.

Sounds contrived, but if you start with the assumption that in `quickcheck` random is the default, it make sense that slightly more complex expressions be necessary to express determinism. Another simple constructor is `select` which creates a generator that picks randomly from a list, provided as argument – not unlike `sample`, but consistent with the `quickcheck` definition of generator.

```
select(1:5)()
```

```
[1] 4
```

```
select(1:5)()
```

```
[1] 4
```

The default distributions for all the generators were picked based on implementation convenience more than anything and will be refined in future releases and based on feedback.

Advanced topics

Composition of generators

The alert reader may have already noticed how generators can be used to define other generators. For instance, a random list of double vectors can be generated with `rlist(rdouble)` and a list thereof with `rlist(function() rlist(rdouble))`. Since typing `function()` over and over again gets old quickly and adds clutter, we can use `~` as a shortcut `rlist(~rlist(rdouble))`.

Custom generators

There is no reason to limit oneself to built-in generators and one can do much more than just change the parameters. For instance, we may want to make sure that extremes of the allowed range are hit more often than the built-in generators ensure. For instance, `rdouble` uses by default a standard normal, and values like 0 and Inf have very small or 0 probability of occurring. Let's say we want to test the following assertion about the ratio:

```
is.reciprocal.self.inverse = function(x) isTRUE(all.equal(x, 1/(1/x)))
```

We can have two separate tests, one for values returned by `rdouble`:

```
test(function(x = rdouble()) is.reciprocal.self.inverse(x))
```

```
Pass  function (x = rdouble())
 is.reciprocal.self.inverse(x)
```

```
[1] TRUE
```

and one for the corner cases:

```
test(function(x = select(c(0, -Inf, Inf))()) is.reciprocal.self.inverse(x))
```

```
Pass  function (x = select(c(0, -Inf, Inf))())
 is.reciprocal.self.inverse(x)
```

```
[1] TRUE
```

That's a start, but the two type of values never mix in the same vector. We can combine the two with a custom generator

```
rdoublex =
  function(element = 100, size = 10) {
        data = rdouble(element, size)
        sample(
            c(data, c(0, -Inf, Inf)),
            size = length(data),
            replace = FALSE)}
rdoublex()
```

```
 [1] 100.91898  99.95507 101.12493 100.82122 100.73832  99.37876      -Inf
 [8]  97.78530 100.38984  99.98381 101.51178 100.57578 100.59390 100.94384
```

```
rdoublex()
```

```
[1]  98.62294   0.00000  99.60571 100.38767       Inf  99.58501  99.89721
[8]  99.94619
```

And use it in a more general test.

```
test(function(x = rdoublex()) is.reciprocal.self.inverse(x))
```

```
Pass  function (x = rdoublex())
 is.reciprocal.self.inverse(x)
```

```
[1] TRUE
```