Property-based testing: an introduction

Max Vistrup, based on Ken's slides from last year

October 4, 2022

Property-based testing: the basics

Unit testing and its pitfalls

- The most famous testing paradigm is **unit testing**, where the programmer will manually write test cases.
- For example, a test could assert that for a function `factorial`, a certain input 3 gives a certain output 5.
- Unfortunately, writing many test cases can become repetitive and unwieldy, especially if the participating data structures are large, complex, and annoying to instantiate by hand.
- Unit tests can also be hard to read and understand, as they consist of a set of example input and expected output, but no formal explanation as to why the test is to be expected to succeed.

An alternative: property-based testing

An alternative to unit testing is **property-based testing**, where the programmer will write down a list of *properties* that functions should satisfy, and these will be automatically tested against *random input* in search of a *counterexample*.

Slogan: instead of writing tests by hand, we generate them.

Example of properties

Suppose we have defined operations for addition and multiplication:

```
(+) :: Int -> Int -> Int
(*) :: Int -> Int -> Int
```

Then we could state properties that we expect them to hold, for example

- *(Commutative property for addition.)* $\forall x.\forall y.x + y = y + x$
- *(Distributive property.)* $\forall x.\forall y.\forall z.(x + y) * z = x * z + y * z$
- ...

If we can find values for x, y, z for which one of these properties fails to hold, then surely the code we have written must be wrong, i.e., the test fails!

QuickCheck in Haskell

Haskell lends itself to property-based testing in view of its lack of side-effects. The famous **QuickCheck** library implements property-based testing for Haskell. Today, we introduce the basics of its API.

A first peek at QuickCheck

Consider our two properties of $+$ and $*$ stated earlier. How would we verify them with QuickCheck?

```haskell
import Test.QuickCheck

prop_CommutativeAddition :: Int -> Int -> Property
prop_CommutativeAddition x y = x + y === y + x

prop_Distributive :: Int -> Int -> Int -> Property
prop_Distributive x y z = (x + y) * z === x * z + y * z
```

Let us run the tests:

```
ghci> quickCheck prop_Distributive
+++ OK, passed 100 tests.
ghci> quickCheck prop_CommutativeAddition
+++ OK, passed 100 tests.
```

In each case, 100 random examples are chosen to check the properties we have defined.

This is not a proof that the properties hold, but it is *emperical evidence*.

A failing property

Let's see what happens if we write down a wrong property.

```haskell
import Test.QuickCheck

prop_AssocFloat :: Double
                -> Double
                -> Double
                -> Property
prop_AssocFloat x y z = (x + y) + z === x + (y + z)
```

```
ghci> quickCheck prop_AssocFloat
*** Failed! Falsified (after 6 tests and 6 shrinks):
-1.0
0.1
1.0
9.999999999999998e-2 /= 0.10000000000000009
```

The last line is the failing equality. The three lines preceding it are the values for the arguments (respecitvely x, y, z) that constitute the counterexample.

QuickCheck fundamentals

With a few examples in mind, we now dive into the workings of QuickCheck.

- A `Property` carry instructions for how to generate random test cases, how to test them, and what to print.
- A `Gen a` (called a generator) specifies how to randomly pick values of type `a`.
  - → One can extract random values from a generator using the `generate` function. This is useful for testing your generators.

Working with generators

- `Gen` is a `Monad` in the natural way. In particular, `return x` is the generator that always yields `x`.
- The `Arbitrary` type class is used to specify how to randomly generate arbitrary values of a type.

```
class Arbitrary a where
    arbitrary :: Gen a
    -- ...
```

→ `Arbitrary` is implemented for many basic types. Therefore, one can use `arbitrary` to get many simple generators. For example, to flip a coin, we can do

```
do x <- arbitrary :: Gen Bool
   if x then return "heads"
        else return "tails"
```

- `chooseInt (a, b)` is the generator that picks out an integer between `a` and `b` (inclusive).
- `elements :: [a] -> Gen a` produces the generator that picks out a random element of the list.
- `oneof :: [Gen a] -> Gen a` builds a generator that generates using a generator chosen at random from the list.
  - → For example, `oneof [return 1, chooseInt (10, 20)]` is the generator that with probability $\frac{1}{2}$ generates 1 and otherwise picks a random integer in the range from 10 to 20.

- `frequency :: [(Int, Gen a)] -> Gen a` is a variant of `oneof` that gives you control over probabilities.
  - → For example, `frequency [(1, return 5), (2, return 7)]` yields 5 with probability $\frac{1}{3}$ and 7 with probability $\frac{2}{3}$.
- `vectorOf :: Int -> Gen a -> Gen [a]` produces a generator for a list of `n` items.
  - → For example, `vectorOf 3 (elements [1, 2])` produces lists like `[2, 1, 2]` and `[2, 2, 2]`.

The `quickCheck` function

```haskell
quickCheck :: Testable prop
           => prop
           -> IO ()
```

where `Testable` is a type class:

```haskell
class Testable prop where
    property :: prop -> Property

instance (Arbitrary a, Show a, Testable prop)
        => Testable (a -> prop) where
    property f = forAll arbitrary f
```

Take-away: Function arguments represent universal quantifiers.

An example: Morse code

Encoding and decoding Morse code

```haskell
table :: [(Char, String)]
table = [('A', ".-"), ('N', "-."), ...]
encode :: String -> String
encode [] = []
encode (x:xs) = fromJust (lookup x table) ++ encode xs
decode :: String -> [String]
decode "" = [""]
decode m = concatMap
    (\(letter, code) ->
        case stripPrefix code m of
            Just rest -> map (letter:) (decode rest)
            Nothing -> []
    )
    table
```

Properties of Morse code

We certainly expect the following property to hold for our program:

(∗) Encoding a string `str` and decoding it again yields `str` and perhaps some other strings.

Indeed, remember that Morse code is ambigious, which is also why `decode` returns a list.

We may test $(*)$ as follows:

```haskell
import Test.QuickCheck

-- Note that 'Bool' implements 'Testable', so this
-- is 'quickCheck'-able.
prop_EncodeDecode :: String -> Bool
prop_EncodeDecode str = str `elem` decode (encode str)
```

Let's try to run it:

```
ghci> quickCheck prop_EncodeDecode
*** Failed! (after 2 tests and 1 shrink):
Exception:
  Maybe.fromJust: Nothing
  CallStack (from HasCallStack):
    error, called at libraries/base/Data/Maybe.hs:149:21
                     in base:Data.Maybe
    fromJust, called at test/Test.hs:60:17 in main:Main
"a"
```

The issue: `encode` doesn't accept arbitrary strings, only all uppercase alphabetic strings.

We have to make a custom generator:

```haskell
-- | A string of alphabetic, uppercase characters.
data UpperCaseString = UpperCaseString String

instance Arbitrary UpperCaseString where
    arbitrary = do
        n <- chooseInt (0, 5)
        str <- vectorOf n (elements ['A'..'Z'])
        return $ UpperCaseString str

prop_EncodeDecode :: UpperCaseString -> Bool
prop_EncodeDecode (UpperCaseString str) =
    str `elem` decode (encode str)
```

Let's try again:

```
ghci> quickCheck prop_EncodeDecode
=== prop_EncodeDecode from test/Test.hs:84 ===
+++ OK, passed 100 tests.
```

Testing Algebraic Data Types

An example

```haskell
data Expr = Cst Int
          | Add Expr Expr
    deriving (Eq, Show)

eval :: Expr -> Int
eval (Cst n) = n
eval (Add x y) = eval x + eval y

prop_CommutativeAddition :: Expr -> Expr -> Property
prop_CommutativeAddition x y =
    eval (Add x y) === eval (Add y x)
```

Incomplete: We need to implement `Arbitrary` for `Expr`.

Generating ASTs: the naïve way

```
instance Arbitrary Expr where
    arbitrary = oneof [ fmap Cst arbitrary
                      , do x <- arbitrary
                           y <- arbitrary
                           return $ Add x y
                      ]
```

Issue: This can generate huge expressions!

Generating ASTs: the good way

Solution: Sized generators.

```haskell
-- | A generator for 'Expr's with an upper bound on
--   approximately the number of nodes.
exprN :: Int -> Gen Expr
exprN 0 = fmap Cst arbitrary
exprN n = oneof
    [ fmap Cst arbitrary
    , do x <- exprN (n `div` 2)
         y <- exprN (n `div` 2)
         return $ Add x y
    ]
instance Arbitrary Expr where
    arbitrary = sized exprN
```

QuickCheck Mini for Erlang

QuickCheck Mini

We will use Quviq QuickCheck Mini for Erlang in this course.

You will need to install it on your computer. A guide is available on Absalon.

Dictionaries

The `dict` module in Erlang implements a purely functional hash table.

- `dict:new()` creates an empty dictionary.
- `dict:store(Key, Value, OldDict)` creates a new dictionary from `OldDict` where `Value` is stored at `Key`.
- `dict:store(Key, Dict)` retrieves a value from a dictionary.
- ... (see documentation)

Let us write some tests for `dict` using QuickCheck Mini.

Generating dictionaries

```
key() ->
    eqc_gen:oneof([atom(), eqc_gen:int(), eqc_gen:real()]).
value() ->
    eqc_gen:oneof([eqc_gen:int(), atom()]).
atom() ->
    eqc_gen:elements([a,b,c,d]).
dict_0() ->
    ?LAZY(
        eqc_gen:oneof([
            dict:new(),
            ?LET({K,V,D}, {key(), value(), dict_0()},
                 dict:store(K,V,D))
        ])
    ).
```

Some analogies to QuickCheck for Haskell

- `key()`, `value()`, and `atom()` in above example are generators in the same sense as QuickCheck for Haskell.
- `?LET(v, gen(), expr)` should be thought of as equivalent to

  ```
  do v <- gen
     expr
  ```
- Non-generators are for the most part automatically coerced into generators with constant value. This is unlike QuickCheck for Haskell, where you have to manually use `return`.
- `?LAZY()` defers execution of code describing a generator. Without it in the `dict` example, we would get a stack overflow because Erlang has eager evaluation.

Testing some properties

```
no_duplicates(Lst) ->
    length(Lst) =:= length(lists:usort(Lst)).

prop_unique_keys() ->
    ?FORALL(D, dict_0(),
            no_duplicates(dict:fetch_keys(D))).
```

Let's try to run our test:

```
> eqc:quickcheck(d:prop_unique_keys()).
...................................Failed! After 67 tests.
{dict,8,16,16,8,80,48,
  {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
  {{[[-2.0|-13]], [],
    [[b|d]], [[c|c],[3.2|-6]], [[d|7]],
    [],[],[],[],[], [[-2|-4],[14|b]],
    [],[], [[-9|12]], [],[]}}}
Shrinking xxxx.xx.x.xxxxxxxxxxxxxxx.xxxxxxxxxxx(5 times)
{dict,2,16,16,8,80,48,
  {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
  {{[[-2.0|0]],[],[],[],[],[],[],[],[],[],
    [[-2|0]],[],[],[],[],[]}}}
false
```

Symbolic testing

Issue: the counterexample is complete nonsense.

- To get more readable counterexamples, it sometimes makes sense to construct random structures symbolically.
- In Erlang, we might represent a call as `{call, Module, Function, ArgumentList}`
  - → Such representations of symbolic calls can be evaluated with the `eqc_symbolic:eval` function.
- Examples:

Function call	**Symbolic call**
`dict:new()`	`{call, dict, new, []}`
`dict:store(K,V,D)`	`{call, dict, store, [K,V,D]}`
`lists:usort([3,2])`	`{call, list, usort, [[3,2]]}`

Symbolic testing for dict

```
dict_1() ->
    ?LAZY(
       oneof([{call,dict,new,[]},
              ?LET(D, dict_1(),
               {call,dict,store,[key(),value(),D]})])
      ).

prop_unique_keys() ->
    ?FORALL(D,dict_1(),
            no_duplicates(dict:fetch_keys(
                eqc_symbolic:eval(D))))).
```

Testing the quality of your generator

```
prop_measure() ->
    ?FORALL(D,dict(),
        collect(length(dict:fetch_keys(eval(D))),true)).

prop_aggregate() ->
    ?FORALL(D,dict(),
        aggregate(eqc_symbolic:call_names(D), true)).
```

```
> eqc:quickcheck(d:prop_measure()).
....................................................
OK, passed 100 tests
54% 0
28% 1
8% 2
7% 3
1% 12
1% 6
1% 4
true
```

```
> eqc:quickcheck(d:prop_aggregate()).
.......................................................
OK, passed 100 tests
50.5% {dict,store,3}
49.5% {dict,new,0}
true
```

More improvements

- Statistics indicate that we often generate very small or empty dictionaries. Use `eqc_gen:frequency` to attain finer control over generation:

```
dict_2() ->
    ?LAZY(
      eqc_gen:frequency([{1,{call,dict,new,[]}},
          {4,?LET(D, dict_2(),
            {call,dict,store,[key(),value(),D]})}])
    ).
```

- Better counterexamples with shrinking:

```
dict_3() ->
    ?LAZY(
       eqc_gen:frequency([{1,{call,dict,new,[]}},
              {4,?LETSHRINK([D],[dict_3()],
                {call,dict,store,[key(),value(),D]})}])
      ).
```

→ `?LETSHRINK([A, B, ...], [f1(), f2(), ...], expr)` is similar to `?LET([A, B, ...], [f1(), f2(), ...], expr)` but will try to "shrink" a counterexample to one of `A`, `B`, ... Next lecture will have more on shrinking.

Returning to our property

```
no_duplicates(Lst) ->
    length(Lst) =:= length(lists:usort(Lst)).

prop_unique_keys() ->
    ?FORALL(D,dict_3(),
            no_duplicates(dict:fetch_keys(eval(D)))).
```

What is the problem?

```
> eqc:quickcheck(d:prop_unique_keys()).
Failed! After 1 tests.
{call,dict,store,
  [0.0,b,{call,dict,store,[0,c,{call,dict,store,
  [0,0,{call,dict,new,[]}]}]}]}
Shrinking ..(2 times)
{call,dict,store,
  [0.0,a,{call,dict,store,[0,a,{call,dict,store,
  [0,0,{call,dict,new,[]}]}]}]}
false
```

In the `dict` module, two keys are different if they don't match according to `=:=`. However, `lists:usort/1` function uses `==` for equality.

```
> 1 =:= 1.0.
false
> 1 == 1.0.
true
```

Summary

- Instead of writing a lot of unit tests, write down a number of properties for your code and test them on random inputs.
- The QuickCheck library for Haskell (and QuickCheck Mini for Erlang) streamlines the process.
- To generate random data, you have to write so-called *generators*.
- Generators are usually built out of existing generators by use of the monad structure and other provided functions.