Advanced Programming

Property-based Testing – Introduction

Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

October, 2021

Part I

Pre-lecture – QuickCheck Introduction

Testing – Challenges with Unit-testing

- Testing is a cost-effective way to help us assess the correctness of our code. However, writing test cases are boring (and sometimes hard).
- Unit tests can be hugely beneficial when developing, even as part of the development process. Writing a few tests can do wonders for your understanding of the problem domain.
- Writing many test-cases quickly get repetitive.
- Ultimately we end up with *a set of examples* of expected result from some given inputs. But with *no specification* telling us why we should expect those results.

QuickCheck in One Slide

- Property-based testing is a way to test against a specification, rather than a set of examples.
- If it is repetitive and hard to come up with good input data, we should automate it.
  We should *generate* random input data (from a suitable distribution).
- Often we write many test for the same underlying idea — instead, write down that underlying idea (property) and *generate* test cases from that.
- QuickCheck motto: don't write a unit test-suite – *generate* it.

Properties for Some Well-known Function

- Plus distribute over *max*:
  $\forall k\ x\ y.\ k + max(x, y) = max(k + x, k + y)$
- Plus is associative:
  $\forall x\ y\ z.\ x + (y + z) = (x + y) + z$
- We now write these as Haskell function using the QuickCheck library/EDSL:

```haskell
import Test.QuickCheck

prop_plusmax :: Int -> Int -> Int -> Property
prop_plusmax k x y =
  k + (max x y) === max (k + x) (k + y)

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

Testing The Properties

```
> quickCheck prop_plusmax ()
+++ OK, passed 100 tests. ()
> verboseCheck prop_int_plus_associative ()
Passed:
0
0
0
0 == 0
...
Passed:
38
-67
82
53 == 53

+++ OK, passed 100 tests.
```

Property for List Function

- QuickCheck also works for data-structures
- For list reversal we have the property:
  $\forall xs.\mathtt{reverse}(\mathtt{reverse}(xs)) = xs$
- As code:

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

- Checking:

```
> quickCheck prop_reverse
+++ OK, passed 100 tests.
```

Some Non-Properties

- Let's try something that'll fail.
  For instance, that minus distribute over max, or that plus is associative for floating point numbers
- As code:

```
prop_minusmax :: Int -> Int -> Int -> Property
prop_minusmax k x y =
  k - (max x y) === max (k - x) (k - y)


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

Finding Failures

```
> quickCheck prop_minusmax
*** Failed! Falsified (after 3 tests and 5 shrinks):
0
0
1
-1 /= 0 ()
> quickCheck prop_double_plus_associative
*** Failed! Falsified (after 2 tests and 6 shrinks):
-0.1
0.3
0.5
0.7000000000000001 /= 0.7
```

QuickCheck in Erlang

- For Erlang we'll use Quviq Erlang QuickCheck Mini (see installation instructions on Absalon)
  Another popular alternative is PropEr.
- In Haskell we use overloading via type-classes to make a nice library for writing tests.
- In Erlang we use macros to make a nice library for writing tests.

The Same Properties in Erlang – The Good Ones

```
-include_lib("eqc/include/eqc.hrl").

prop_plusmax() ->
  ?FORALL({K, X, Y}, {int(), int(), int()},
          eqc:equals(K + max(X, Y),
                     max(K + X, K + Y))).

prop_int_plus_associative() ->
  ?FORALL({X, Y, Z}, {int(), int(), int()},
          eqc:equals(X + (Y + Z),
                     (X + Y) + Z)).

prop_reverse() ->
  ?FORALL(XS, list(int()),
         eqc:equals(lists:reverse(lists:reverse(XS)),
                    XS)).
```

The Same Properties in Erlang – The Bad Ones

```
prop_minusmax() ->
  ?FORALL({K, X, Y}, {int(), int(), int()},
          eqc:equals(K - max(X, Y),
                     max(K - X, K - Y))).

prop_real_plus_associative() ->
  ?FORALL({X, Y, Z}, {real(), real(), real()},
          eqc:equals(X + (Y + Z),
                     (X + Y) + Z)).
```

Testing the Properties

```
2> eqc:quickcheck(intro:prop_plusmax()).
eqc:quickcheck(intro:prop_plusmax()).
..................................................................
OK, passed 100 tests

3> eqc:quickcheck(intro:prop_real_plus_associative()).
eqc:quickcheck(intro:prop_real_plus_associative()).
.....................Failed! After 22 tests.
{-1.75,-1.0,-0.3333333333333333}
-3.083333333333333 /= -3.0833333333333335
Shrinking xx...x..xxx..xxxx.xxxx.xxx.xxxx.x.xx.xxxxx.xxxx(14 tim
{1.0,1.0,0.3333333333333333}
2.333333333333333 /= 2.3333333333333335
false
```

Today's Program

- Property based testing
- QuickCheck building blocks
- QuickCheck in Haskell
- QuickCheck in Erlang

Part II

Morse Code

Morse Code

- One of the exercises from exercise set 1 is about decoding morse code.
- That is, write the functions

  ```
  encode :: String -> String
  decode :: String -> [String]
  ```

  For instance, "`...---..-....-`" could be the encoding for both Sofia and Eugenia.

Morse Code, Implementation

```haskell
import qualified Data.List as L

charMap = [('A', ".-"), ('B', "-..."), ... ]

findChar c = fromMaybe "" $ L.lookup c charMap

encode :: String -> String
encode = concatMap findChar
decode :: String -> [String]
decode ""    = [""]
decode input = [ c : rest | (c, code) <- charMap
                          , code `L.isPrefixOf` input
                          , let clen = length code
                          , rest <- decode $ drop clen input]
```

Unit Testing In Haskell

```
import Test.Tasty
import Test.Tasty.HUnit
import Morse

testsuite =
  testGroup "Testing Morse encoding decoding"
  [ testGroup "Encoding"
    [ testCase "Encode SOFIA (testing with @=?)"
      ("...---..-....-" @=? encode("SOFIA"))
    , testCase "Encode EUGENIA"
      ("...---..-....-" @=? encode("EUGENIA")) ]
  , testGroup "Decoding"
    [ testCase "Decode Eugenia"
      (assertBool "EUGENIA is not in decodings of ...---..-....-"
       ("EUGENIA" `elem` decode "...---..-....-"))]
  ]
main = defaultMain testsuite
```

Part III

Property Based Testing

QuickCheck in One Slide

- Property-based testing is a way to test against a specification, rather than a set of examples.
- If it is repetitive and hard to come up with good input data, we should automate it.
  We should *generate* random input data (from a suitable distribution).
- Often we write many test for the same underlying idea — instead, write down that underlying idea (property) and *generate* test cases from that.
- QuickCheck motto: don't write a unit test-suite – *generate* it.

Property Based Testing

- What is a good property to test for the `Morse` code library?
- For the `Morse` module we would expect to be able to decode an encoded string:

$$\forall s.\ s = decode(encode(s))$$

- Alas, that's too strong a property. Several strings can have the same encoding. Thus the property we are after is

$$\forall s.\ s \in decode(encode(s))$$

QuickCheck Property for Morse Code

We write our property as a predicate, which we can then test with the function `quickCheck`

```
import Test.QuickCheck
import qualified Morse

prop_encode_decode s = s `elem` Morse.decode (Morse.encode s)
```

QuickCheck Building Blocks

- QuickCheck is a (EDSL) library for specifying two main data-types: `Property` and `Gen a`
- `Property` for specifying properties, actually uses the `Testable prop` type-class
  Some interesting instances:

  ```
  Testable Bool
  (Arbitrary a, Show a, Testable prop) => Testable (a -> prop)
  ```

  From something `Testable` we can generate *test cases*
- `Gen a` is a generator for values of type `a`, often by using the `Arbitrary a` type class.

QuickCheck Generator Building Blocks

- QuickCheck generates random values by clever use of the `Arbitrary` type-class:

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

- That uses the type:

  ```
  newtype Gen a = MkGen { unGen :: QCGen -> Int -> a }
  ```

  to generate values of type `a`.

- `Gen` is a monad.

- Functions for combining generators (combinators). E.g.:

  ```
  choose :: Random a => (a, a) -> Gen a
  oneof :: [Gen a] -> Gen a
  listOf :: Gen a -> Gen [a]
  vectorOf :: Int -> Gen a -> Gen [a]
  ```

QuickCheck for Morse, Strings with Only Letters

```haskell
import Test.QuickCheck
import qualified Data.Char as C
import qualified Morse

upper = map C.toUpper
prop_encode_decode (LO s) = upper s `elem`
                              Morse.decode (Morse.encode s)

asciiLetter = elements (['a'..'z'] ++ ['A'..'Z'])

newtype LettersOnly = LO String
                      deriving (Eq, Show)

instance Arbitrary LettersOnly where
  arbitrary = fmap LO (listOf asciiLetter)
```

QuickCheck for Morse, Limit Length

```
import Test.QuickCheck
import qualified Data.Char as C
import qualified Morse

upper = map C.toUpper
prop_encode_decode (LO s) = upper s `elem`
                            Morse.decode (Morse.encode s)

asciiLetter = elements (['a'..'z'] ++ ['A'..'Z'])

newtype LettersOnly = LO String
                      deriving (Eq, Show)

instance Arbitrary LettersOnly where
  arbitrary = fmap LO $ do
    n <- choose (0, 4)
    vectorOf n asciiLetter
```

QuickCheck for Morse, Character Frequencies

```haskell
import Test.QuickCheck
import qualified Data.Char as C
import qualified Morse

upper = map C.toUpper
prop_encode_decode (LO s) = upper s `elem`
                               Morse.decode (Morse.encode s)

weightedLetters = frequency [(2 ^ (max - length code), return c)
                            | (c,code) <- Morse.charMap]
  where max = 1 + (maximum $ map (length . snd) Morse.charMap)

newtype LettersOnly = LO String deriving (Eq, Show)

instance Arbitrary LettersOnly where
  arbitrary = fmap LO $ do n <- choose (0, 7)
                           vectorOf n weightedLetters
```

Part IV

Testing Algebraic Data Types

Testing Algebraic Data Types

How can be generate random expressions for checking that `Add` is commutative:

```
data Expr = Con Int
          | Add Expr Expr
     deriving (Eq, Show, Read, Ord)

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

prop_com_add x y = eval (Add x y) === eval (Add y x)
```

Generating Exprs

- Our first attempt

```haskell
expr =  oneof [ fmap Con arbitrary
              , do x <- expr
                   y <- expr
                   return $ Add x y]

instance Arbitrary Expr where
  arbitrary = expr
```

  is correct,

- … but may generate humongous expressions.
- Instead we should generate a `sized` expression

```haskell
expr = sized exprN
exprN 0 = fmap Con arbitrary
exprN n = oneof [ fmap Con arbitrary
                , do x <- subexpr
                     y <- subexpr
                     return $ Add x y]
  where subexpr = exprN (n `div` 2)
```

Part V

Testing Libraries – In Erlang

Dictionaries

- `dict`: purely functional key-value store
  - `new()`
  - `store(Key,Value,Dict)`
  - `fetch(Key, Dict)`
  - ...

Internal Representation

- ```
  > D1 = dict:store(1, a, dict:new()).
  {dict,1,16,16,8,80,48,
    {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
    {{[],[],[],[],[],[],[],[],[],[],[],[[1|a]],[],[],[],[]}}}
  > D2 = dict:from_list([{a,1}, {b,2}, {1,abba}]).
  {dict,3,16,16,8,80,48,
    {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
    {{[], [[a|1]], [[b|2]],
      [],[],[],[],[],[],[],[],
      [[1|abba]], [],[],[],[]}}}
  ```

- “No, stop! Don’t expose your `dict`”
  - Complex representation
  - Complex invariants
  - We’ll just test the API
  ```

Keys Should Be Unique

- There should be no duplicate keys

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

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

We need a generator for dicts

Generating dicts

- Generate `dicts` using the API

```
key() ->
    oneof([atom(), int(), real()]).
value() ->
    oneof([int(), atom()]).
atom() ->
    elements([a,b,c,d]).

dict_fl() ->
  ?LET(X, list({key(), value()}), dict:from_list(X)).

dict_0() ->
    ?LAZY(
       oneof([dict:new(),
              ?LET({K,V,D},{key(), value(), dict_0()},
                   dict:store(K,V,D))])
      ).
```

How did I get here?

```
> 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 Calls

- Rather than executing a call straight away in a generator, build a data structure with the calls.
- A call is represented as:

  `{call, Module, Function, ArgumentList}`
- Examples:

Function call	Symbolic call
`dict:new()`	`{call, dict, new, []}`
`dict:store(K,V,D)`	`{call, dict, store, [K,V,D]}`
`lists:usort([3,2,1])`	`{call, lists, usort, [[3,2,1]]}`
`myfun(X, Y)`	`{call, ?MODULE, myfun, [X, Y]}`

Generate `dicts` Symbolically

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

Properties for Symbolic Values

A symbolic dict most be evaluated before use

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

```
> 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
```

How good is our generator

- Some properties for measuring the quality of a generator

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

prop_aggregate(DGen) ->
    ?FORALL(D, DGen(),
            aggregate(call_names(D), true)).
```

```
> eqc:quickcheck(d:prop_measure(fun d:dict_1/0)).
...................................................
OK, passed 100 tests

54% 0
28% 1
8% 2
7% 3
1% 12
1% 6
1% 4
true
```

```
> eqc:quickcheck(d:prop_aggregate(fun d:dict_1/0)).
......................................................
OK, passed 100 tests

50.5% {dict,store,3}
49.5% {dict,new,0}
true
```

More Improvements

- We can use `frequency` to generate more interesting `dict`s

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

- I need a shrink now

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

```
> eqc:quickcheck(d:prop_measure(fun d:dict_3/0)).
.................................... OK, passed 100 tests

18% 0
15% 3
15% 2
14% 1
...
2% 12
1% 11
true
> eqc:quickcheck(d:prop_aggregate(fun d:dict_3/0)).
.................................... OK, passed 100 tests

79.3% {dict,store,3}
20.7% {dict,new,0}
true
```

```
> eqc:quickcheck(d:prop_unique_keys()).
.......Failed! After 8 tests.
{call,dict,store,
 [1,a, {call,dict,store, [-1.0,a, {call,dict,store, [-2.0,b,
  {call,dict,store, [0,a, {call,dict,store, [1.0,2,
   {call,dict,store, [a,0, {call,dict,store, [-2,d,
    {call,dict,store, [0,-2, {call,dict,store, [-1.0,d,
     {call,dict,store, [1,0, {call,dict,store,[d,a,
      {call,dict,store, [1,b, {call,dict,store, [-1,1,
       {call,dict,store, [0,2, {call,dict,store, [d,2,
        {call,dict,new,[]}]}]}]}]}]}]}]}]}]}]}]}]}]}]}]}]}]}]}]}
Shrinking .........x..x...(15 times)
{call,dict,store,[0,a,
 {call,dict,store,[0.0,0,{call,dict,new,[]}]}]}
false
```

Back To The Property

- What is the problem with out property:

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

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

- In the `dict` module two keys are different if they do not match `=:=` The `lists:usort/1` function uses `==` for equality.

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

Part VI

Summary

Summary

- Use QuickCheck for better testing
- Property-based testing:
  - Identify properties to test
  - Write data-generators (in Haskell using `Arbitrary` and `Gen`)
  - When a pro: shrinking
- Be careful with your specification
- We can test complex data structures by generating sequences of API calls, preferbly with symbolic commands
- Remember to measure the quality of your generators