Functional and Logic Programming

Lecture 12 — Property based testing

Pedro Vasconcelos
DCC/FCUP

2025

Contents

Overview

Unit tests

Property-based tests

Thinking about properties

Reflections

Contents

Overview

Unit tests

Property-based tests

Thinking about properties

Reflections

Overview

- A longstanding challenge in software engineering is to ensure the software behaves correctly
- Formal verification is (still) costly and used only in high-assurance domains
- Tests are the most common form of verification used in practice

Property based testing

- A testing methodology that automatically generates test cases and check them against user-specified properties
- A “lightweight formal method” i.e. one that can be applied by people that don’t fully understand it
- Originated in the functional programming community (the *QuickCheck* library for Haskell)
- But has also been adapted for other paradigms and languages

Contents

Overview

Unit tests

Property-based tests

Thinking about properties

Reflections

Unit tests

- Code fragments for testing functions, classes, libraries, etc.
- Express the expected results for specific inputs
- Example: let us test a function

```
isqrt :: Integer -> Integer
isqrt x = floor (sqrt (fromInteger x))
```

  for computing the integer square root

- Examples:

```
>>> isqrt 2
1
>>> isqrt 3
1
>>> isqrt 4
2
>>> isqrt 9
3
```

Unit tests (cont.)

```
import Control.Exception (assert)

unitTests :: IO ()
unitTests = do
  assert (isqrt 2 == 1) $ putStrLn "test 2"
  assert (isqrt 3 == 1) $ putStrLn "test 3"
  assert (isqrt 4 == 2) $ putStrLn "test 4"
  assert (isqrt 9 == 3) $ putStrLn "test 9"
```

Problems with unit tests

Cognitive bias:

- how can we include an edge case in the tests that we didn't consider in the code?

Poor scaling:

- a few unit tests per function
- for n functions, $O(n)$ unit tests
- but testing *interactions* between functions requires $O(n^2), O(n^3), \ldots$ unit tests

Problems with unit tests

Cognitive bias:

- how can we include an edge case in the tests that we didn't consider in the code?

Poor scaling:

- a few unit tests per function
- for n functions, $O(n)$ unit tests
- but testing *interactions* between functions requires $O(n^2), O(n^3), \ldots$ unit tests

Solution:

> *"Don't write tests — generate them!"*

John Hughes, co-author of *QuickCheck*

Contents

Overview

Unit tests

Property-based tests

Thinking about properties

Reflections

Property-based testing

- Write properties:
  - functions that return a boolean value
  - should hold for all argument values
  - should define the expected behaviour in all cases
- Use generators to supply random test inputs
- The testing framework checks the property
  - testing fails if a counter-example is found
  - otherwise, testing succeeds

Example

Consider the following property of the reverse function: using `reverse` twice should give back the original list.

```
import Test.QuickCheck

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

- Convention: property names start with `prop_`
- The argument is quantified implicitly:
  for all `xs`, `reverse (reverse xs) == xs`

Example (cont.)

We can test interactively in GHCi.

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

- QuickCheck generated 100 random lists
- The equality holds in all test cases

When a property fails

```
prop_rev_wrong :: [Int] -> Bool
prop_rev_wrong xs = reverse xs == xs

>>> quickCheck prop_rev_wrong
*** Failed! Falsified (after 3 tests and 3 shrinks)
[0,1]
```

- QuickCheck produces a conter-example
- `[0,1]` is a counter-example because

  ```
  reverse [0,1] /= [0,1]
  ```

- The counter-example is minimal (more about this later)

Integer square root

Let us go back to the integer square root function.

```
isqrt :: Integer -> Integer
isqrt x = floor (sqrt (fromInteger x))
```

How can test this using QuickCheck?

Specifications vs. algorithms

Let us start by trying to generalize the unit tests:

```
prop_isqrt n = isqrt n == ????
```

- What should we write in the right hand side?

Specifications vs. algorithms

Let us start by trying to generalize the unit tests:

```
prop_isqrt n = isqrt n == resultOfIsqrt n
```

- What should we write in the right hand side?
- Another implementation of the square root function?

Specifications vs. algorithms

Let us start by trying to generalize the unit tests:

```
prop_isqrt n = isqrt n == resultOfIsqrt n
```

- What should we write in the right hand side?
- Another implementation of the square root function?
- But now the property is as difficult to write as the code under test!

Specifications vs. algorithms

Let us start by trying to generalize the unit tests:

```
prop_isqrt n = isqrt n == resultOfIsqrt n
```

- What should we write in the right hand side?
- Another implementation of the square root function?
- But now the property is as difficult to write as the code under test!
- We need something simpler: a specification rather than an algorithm

  specification: a rigorous definition of valid solutions
  algorithm: a constructive method for obtaining a solution

Specifying the square root

Let $n \geq 0$ be a non-negative number and $r = \texttt{isqrt n}$. Then

$$r \geq 0 \wedge r^2 \leq n \wedge (r+1)^2 > n$$

Hence: r should be the *greatest* non-negative integer such that $r^2 \leq n$.

- This specification defines *exactly* what the result of the square root function should be
- It allows testing if we got the correct solution but it's much simpler than an algorithm

Specifying the square root (cont.)

We can write the specification as a QuickCheck property:

```
prop_isqrt :: Integer -> Bool
prop_isqrt n
    = let r = isqrt n
      in r>=0 && r^2<=n && (r+1)^2>n
```

Testing with QuickCheck

```
>>> quickCheck prop_isqrt
*** Failed! Falsified (after 6 tests and 2 shrinks)
-1
```

- This is not a valid counter-example: `isqrt` only needs to work for non-negative integers
- But the default generator allows for arbitrary integer values
- We need to impose a pre-condition to restrict to the valid cases

Pre-conditions

```
prop_isqrt :: Integer -> Property
prop_isqrt n
  =  n>=0 ==> let r = isqrt n
              in r>=0 && r^2<=n && (r+1)^2>n
```

- The operator `==>` specifies a pre-condition
- When the pre-condition fails, QuickCheck discards the test case and tries again
- This way we ensure that we do not try square roots of negative values
- Note that the type becomes `Property` instead of `Bool`

Testing again

```
>>> quickCheck prop_isqrt
+++ OK, passed 100 tests; 87 discarded.
```

- QuickCheck generated 100 non-negative values and 87 negative ones (discarded)
- This is to be expected: roughly half of the generated integers were negative

Generators

- Generators are used to produce random values
- QuickCheck defines several generators for basic types
- Example:

```
choose :: (Int,Int) -> Gen Int
```

- `Gen a` is the type of generators for values of type *a*
- The `generate :: Gen a -> IO a` function uses a stateful pseudo-random generator:

```
>>> generate (choose (1,10))
6
>>> generate (choose (1,10))
3
```

The Arbitrary class

- The Arbitrary type class defines the default generator for each type

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

- The instances of Arbitrary are parametric e.g.:
  - if there is an instance for `a`, then there is another for `[a]`
  - if there is an instance for `a` and another of `b` then there is another for `(a,b)`
- Try it out:

```
>>> generate (arbitrary :: Gen Int)
>>> generate (arbitrary :: Gen (Bool,Int))
>>> generate (arbitrary :: Gen [Int])
```

- The type signatures are needed to tell which result type you want

Controlling generation

- QuickCheck uses the Arbitrary generators for testing properties
- We can refine types in properties to constraint the generation
- For example:

```
prop_isqrt :: NonNegative Integer -> Bool
prop_isqrt (NonNegative n)
    = let r = isqrt n
      in r>=0 && r^2<=n && (r+1)^2>n
```

  tests only non-negative integers
- This is better than a pre-condition because it avoids discarding tests
- `NonNegative` is one of several newtypes defined in QuickCheck

Another example

Let us test a function to insert values into an ordered list maintaining the ordering.

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys)
    | x<=y = x:y:ys
    | otherwise = y:insert x ys
```

Another example (cont.)

A property that says that if the input list is ordered, then the output list is also ordered.

```
prop_insert_ordered :: Int -> [Int] -> Property
prop_insert_ordered x ys
   = ordered ys ==> ordered (insert x ys)
```

For QuickCheck to be able to generate test cases we need to restrict the type e.g. to integers.

Another example (cont.)

Testing this property doesn't go well:

```
>>> quickCheck prop_insert_ordered
*** Gave up! Passed only 85 tests;
1000 discarded tests.
```

- Most random lists are not ordered
- QuickCheck discards too many test cases
- Moreover: the passing test cases are only very short lists

Another example (cont.)

```
prop_insert_ordered :: Int -> [Int] -> Property
prop_insert_ordered x ys
   = ordered ys ==> collect (length ys) $
                    ordered (insert x ys)

>>> quickCheck prop_insert_ordered
*** Gave up! Passed only 69 tests; 1000 discarded
42% 0
36% 1
13% 2
 9% 3
```

- The probability that the list is ordered decreases very quickly as the length increases
- Hence 91% of the test lists had 2 or fewer elements

Improving generation

Instead of filtering using a pre-condition, we could generate ordered lists using the `OrderedList` modifier.

```
prop_insert_ordered :: Int -> OrderedList Int
                    -> Property
prop_insert_ordered x (Ordered ys)
   = collect (length ys) $ ordered (insert x ys)
```

Improving generation

Instead of filtering using a pre-condition, we could generate ordered lists using the `OrderedList` modifier.

```
prop_insert_ordered :: Int -> OrderedList Int
                    -> Property
prop_insert_ordered x (Ordered ys)
   = collect (length ys) $ ordered (insert x ys)
```

```
>>> quickCheck prop_insert_ordered
+++ OK, passed 100 tests:
 5% 27
 5% 3
 5% 7
 4% 16
...
```

Shrinking

- QuickCheck attempts to simplify counter-examples before presenting them using a process called shrinking
- Shrinking is useful to remove irrelevant details from randomly generated data

Shrinking (cont.)

```
prop_rev_wrong :: [Int] -> Bool
prop_rev_wrong xs
  = reverse xs == xs

>>> quickCheck prop_wrong
*** Failed! Falsified (after 4 tests and 2 shrinks)
[0,1]
```

The counter-example is minimal:

- `[]`, `[0]` or `[1]` are not counter-examples;
- `[0,0]` is not a counter-example.

How shrinking works

- There is an extra `shrink` method in the Arbitrary class

```
class Arbitrary where
   arbitrary :: Gen a
   shrink :: a -> [a]
```

- Given some value, `shrink` should give a list of candidate "simpler" values

How shrinking works (cont.)

The shrinking strategies are type dependent; examples:

Integers shrink towards zero

```
>>> shrink 42
[0,21,32,37,40,41]
```

Tuples shrink by shrinking either of the components

```
>>> shrink (2,3)
[(0,3),(1,3),(2,0),(2,2)]
```

Lists shrink by removing elements or shrinking the elements

```
>>> shrink [2,1]
[[],[1],[2],[0,1],[1,1],[2,0]]
```

How shrinking works (cont.)

- QuickCheck tries replacing an argument value by each shrinking candidate
- If the property still fails, the shrinking process continues
- If there are no candidates or the property passes, QuickCheck stops and reports the smallest counter-example found
- In general, this is only a “local minimum”

Contents

Overview

Unit tests

Property-based tests

Thinking about properties

Reflections

Thinking about properties

- Comming up with properties requires a different mindset than simply thinking about specific examples
- It forces us to rigorously consider what are the fundamental requirements for our code
- It requires abstraction and critical thinking about the problem domain
- At first we may come up with incorrect or incomplete properties
- It is worthwhile taking an adversarial approach:

  What is the worst implementation that still satisfies the specification?

A Hypothetical Persona

The “Enterprise Developer From Hell”[1]:

- Will write code that satisfies all tests
- But will do so in the least helpful way possible!

[1] https://fsharpforfunandprofit.com/

A Hypothetical Dialogue

Could you write a function

```
sort :: [Int] -> [Int]
```

that sorts the values in a list?

A Hypothetical Dialogue

Could you write a function

```
sort :: [Int] -> [Int]
```

that sorts the values in a list?

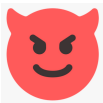
Sure; could you give me some test cases?

A Hypothetical Dialogue

Could you write a function

```
sort :: [Int] -> [Int]
```

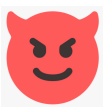
that sorts the values in a list?



Sure; could you give me some test cases?

OK, here is a unit test:

```
test_sort
    = sort [3,2,1] == [1,2,3]
```



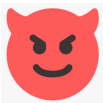
(Goes away and comes back with code).

A Hypothetical Dialogue

Could you write a function

```
sort :: [Int] -> [Int]
```

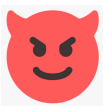
that sorts the values in a list?



Sure; could you give me some test cases?

OK, here is a unit test:

```
test_sort
    = sort [3,2,1] == [1,2,3]
```



(Goes away and comes back with code). Here it is:

```
sort [3,2,1] = [1,2,3]
```

Your function only works for one specific case!

Your function only works for one specific case!

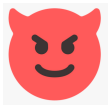
Well, do you have more test cases?

Your function only works for one specific case!

Well, do you have more test cases?

OK, here are some more tests:

```
more_sort_tests
    = and [ sort [3,1,2] == [1,2,3]
          , sort [2,1,1] == [1,1,2]
          ]
```



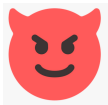
(Goes away and writes more code).

Your function only works for one specific case!

Well, do you have more test cases?

OK, here are some more tests:

```
more_sort_tests
    = and [ sort [3,1,2] == [1,2,3]
          , sort [2,1,1] == [1,1,2]
          ]
```

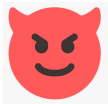


(Goes away and writes more code). OK, got it.

```
sort xs = case xs of
    [3,2,1] -> [1,2,3]
    [3,1,2] -> [1,2,3]
    [2,1,1] -> [1,1,2]
```

(Realizing this is going be more difficult.) Now your function only works for the 3 test cases!

(Realizing this is going be more difficult.) Now your function only works for the 3 test cases!



Well, I simply followed the best “test-driven development” practices. . .

(Realizing this is going be more difficult.) Now your function only works for the 3 test cases!

Well, I simply followed the best “test-driven development” practices. . .

OK, instead of specific tests, I’m giving you a *property* that your code must satisfy.

(Realizing this is going be more difficult.) Now your function only works for the 3 test cases!

Well, I simply followed the best “test-driven development” practices. . .

OK, instead of specific tests, I’m giving you a *property* that your code must satisfy.

```
prop_sort_ordered :: [Int] -> Bool
prop_sort_ordered xs = ordered (sort xs)

ordered xs
     = and (zipWith (<=) xs (tail xs))
```

This says that the output of `sort` must be in ascending order.

(Goes aways and writes some more code.)

(Goes aways and writes some more code.) Here is the revised implementation:

```
sort :: [Int] -> [Int]
sort xs = []
```

But this function is even worse than before; now it only works for the empty list!

(Goes aways and writes some more code.) Here is the revised implementation:

```
sort :: [Int] -> [Int]
sort xs = []
```

But this function is even worse than before; now it only works for the empty list!

The empty list is always ordered; this is the simplest code that will satisfy your requirements.

(Goes aways and writes some more code.) Here is the revised implementation:

```
sort :: [Int] -> [Int]
sort xs = []
```

But this function is even worse than before; now it only works for the empty list!

The empty list is always ordered; this is the simplest code that will satisfy your requirements.

I see what you are doing here. Here's a further property for `sort`.

```
prop_sort_inv_len :: [Int] -> Bool
prop_sort_inv_len xs
    = length (sort xs) == length xs
```

The result list must have the same length as the original list.

 (Goes away for a little while.)

(Goes away for a little while.) No problem; here is a revised implementation:

```
sort :: [Int] -> [Int]
sort xs = replicate (length xs) 0
```

Now the function always returns a list with zeros!

(Goes away for a little while.) No problem; here is a revised implementation:

```
sort :: [Int] -> [Int]
sort xs = replicate (length xs) 0
```

Now the function always returns a list with zeros!

Well, the list with zeros is ordered and has the same length as the input, as specified.

(Goes away for a little while.) No problem; here is a revised implementation:

```
sort :: [Int] -> [Int]
sort xs = replicate (length xs) 0
```

Now the function always returns a list with zeros!

Well, the list with zeros is ordered and has the same length as the input, as specified.

What I meant is that the output list should contain the same values as the input list!

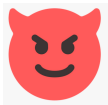
(Goes away for a little while.) No problem; here is a revised implementation:

```
sort :: [Int] -> [Int]
sort xs = replicate (length xs) 0
```

Now the function always returns a list with zeros!

Well, the list with zeros is ordered and has the same length as the input, as specified.

What I meant is that the output list should contain the same values as the input list!



But didn't you specify that. . .

OK, fair enough. I'm adding an extra property: the values in the result list should be a *permutation* of the input list.

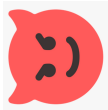
OK, fair enough. I'm adding an extra property: the values in the result list should be a *permutation* of the input list.

```
prop_sort_is_perm :: [Int] -> Bool
prop_sort_is_perm xs
  = sort xs `isPermutation` xs

isPermutation []     ys = null ys
isPermutation (x:xs) ys
  = x`elem`ys &&
    isPermutation xs (delete x ys)
```

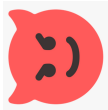
OK, fair enough. I'm adding an extra property: the values in the result list should be a *permutation* of the input list.

```
prop_sort_is_perm :: [Int] -> Bool
prop_sort_is_perm xs
  = sort xs `isPermutation` xs

isPermutation []     ys = null ys
isPermutation (x:xs) ys
  = x`elem`ys &&
    isPermutation xs (delete x ys)
```

By the way, this new property can replace the previous `prop_sort_inv_len`: if two lists are permutations of one another then they must have the same length.



(Defeated.) Well, I guess I have to use a real sorting algorithm. Now, let me look up BogoSort in Haskell...



(Defeated.) Well, I guess I have to use a real sorting algorithm. Now, let me look up BogoSort in Haskell...

The End

Final specification

```
prop_sort_ordered :: [Int] -> Bool
prop_sort_ordered xs = ordered (sort xs)

prop_sort_is_perm :: [Int] -> Bool
prop_sort_is_perm xs
  = sort xs `isPermutation` xs
```

- We need both properties; either one alone is not enough
- This is a complete specification because it corresponds to the *BogoSort algorithm*.

```
sort :: [Int] -> [Int]
sort = head . filter ordered . permutations
```

Contents

Overview

Unit tests

Property-based tests

Thinking about properties

Reflections

Reflections

- Type-class polymorphism would prevent the example-based solutions

```
sort :: Ord a => [a] -> [a]
sort xs = case xs of
    [3,2,1] -> [1,2,3] -- ERROR
    [3,1,2] -> [1,2,3]
    [2,1,1] -> [1,1,2]
```

- But the EDFH could still make up partial solutions:

```
sort :: Ord a => [a] -> [a]
sort [x,y,z]
   | x<=y && y<=z = [x,y,z]
   | y<=x && x<=z = [y,x,z]
   ...
```

Reflections (cont.)

- Writing properties require thinking about the specifications in more abstract and deeper way than unit tests
- QuickCheck gives you an immediate pay-off for writing specifications: you get automatic tests for free
- The specifications are often not obvious; doing automated tests is one way of debugging them
- Writing specifications other benefits: they are a form of machine checked documentation of the code