Property Based Testing : Shrinking Risk In Your Code

Amanda Laucher

@pandamonial
Pariveda Solutions Consultant
Mined Minds Co-Founder

* What is Property Based Testing?
* Patterns for specifying properties
* Generating Inputs to the properties
* Shrinking Failures
* Configurations
* Surprise

YOW! AUSTRALIA

9 Reasons YOU Should Attend YOW! Events

1. **Discover the latest trends**
   Speakers presenting at our events are largely chosen from a network of over 400 authors and experts by an independent, international Program committee based on their recognised expertise as thought leaders in their field. Our international Speakers present by 'invitation only'.

2. **Get a new perspective on your work**
   The relaxed settings for all our events provides the opportunity for attendees to meet and talk with world-renowned speakers face-to-face about your specific development challenges. This is your opportunity to get to know world class speakers on a personal level.

3. **Get unbiased and technically rich talks**
   As former attendees of our events know, each year 40+ world class experts and authors present technically rich talks that don't contain any vendor-hype or marketing spin. Our focus at all our events is on providing good content, not appeasing suppliers..

I <3 types & FP!

I <3 code without bugs!

I don't <3 testing

I don't <3 supporting more code than needed

or religion in software

Bank OCR Kata : Story 2

Account number:

```
3   4   5   8   8   2   8   6   5
```

Position names:

```
d9  d8  d7  d6  d5  d4  d3  d2  d1
```

Checksum calculation:

```
(d1 + 2 + 3*d3 + 4*d4 + 5*d5 + 6*d6 +... 9* d9) % 11 = 0
```

```
describe "#check?" do
  context "when the account number is good" do
    # good account numbers were taken from the user story specs
    Then { checker.check?("000000000").should be_true }
    Then { checker.check?("000000051").should be_true }
    Then { checker.check?("123456789").should be_true }
    Then { checker.check?("200800000").should be_true }
    Then { checker.check?("333393333").should be_true }
    Then { checker.check?("490867715").should be_true }
    Then { checker.check?("664371485").should be_true }
    Then { checker.check?("711111111").should be_true }
    Then { checker.check?("777777177").should be_true }
  end
```

What about these?

2625145141
7634763476
23623762
349745845
2376438X9734
sjdgdfkghfkgsd
bchd
“ “
363729273 65

```
property check  (variable) do
    context "when the account number is good" do
     Then { checker.check?(variable).should be_true }
    end
end


generate_10000000_strings.map(&:check)
```

What do we know about the
correct numbers and incorrect numbers?

Account number:

```
3  4  5  8  8  2  8  6  5
```

Position names:

```
d9 d8 d7 d6 d5 d4 d3 d2 d1
```

Checksum calculation:

```
(d1 + 2 + 3*d3 + 4*d4 + 5*d5 + 6*d6 +... 9* d9) % 11 = 0
```

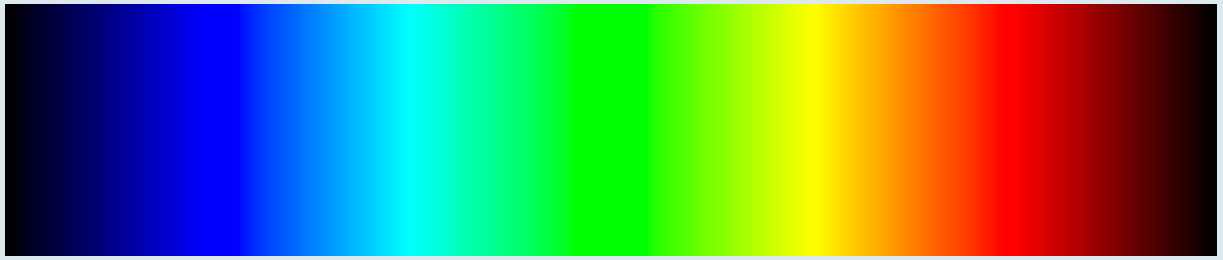
Valid Accounts

"000000000"
“000000051"
"123456789"
“200800000”
“333393333"
“490867715"
“664371485"
"711111111"
"777777177"

* All are 9 chars - anything not 9 chars is invalid
* All are digits
* If we run the check digit multiple times it should return the same value

Property Tests

Unit Tests



Types

"Testing shows the presence, not the absence of bugs"
Edsger W. Dijkstra

Create a model
that specifies **properties**
from functional requirements

Create a model
that specifies properties
from functional requirements

Generate tests
to falsify the properties

Create a model
that specifies properties
from functional requirements

Generate tests
to falsify the properties

Determine minimal failure case

Create a model
that specifies properties
from functional requirements

Generate tests
to falsify the properties

Simple Enough!

Determine minimal failure case

“However learning to Use QuickCheck can be a challenge. It requires the developer to take a step back and really **try to understand** what the code should be doing.”

Property Based Testing With QuickCheck in Erlang

$0.1 + 0.2 + 0.3 = ?$

$$0.1 + 0.2 + 0.3 = ?$$
$$0.3 + 0.2 + 0.1 = ?$$

0.1 + 0.2 + 0.3 = 0.60000000000000001
0.3 + 0.2 + 0.1 = 0.6

A higher level of abstraction
than unit testing

Unit tests:

```
public void testMax() {
  int z = max(1,2)
  assertEquals(2, z)
}
```

Unit tests:

```
public void testMax2() {
  int z = max(1,0)
  assertEquals(1, z)
}
```

Unit tests:

```
public void testMax4() {
  int z = max(-2,0)
  assertEquals(0, z)
}
```

Parameterized Tests

Shift in thinking!

```
property("max") = forAll { (x: Int, y: Int) =>
    z = max(x, y)
    (z == x || z == y) && (z >= x && z >= y)
}
```

```
>OK, passed 100 tests
```

Specify
Generate
Shrink

Specify

Generate

Shrink

forAll: universal quantifier
Test will Pass/Fail

forAll: Claire

```
var commutative =
forAll( _.Int,_.Int ).satisfy( function(a, b) {
    return a + b == b + a }).asTest()
```

forAll: jsverify

```
var boolFnAppliedThrice =
  jsv.forall("bool -> bool", "bool", function (f, b) {
    return f(f(f(b))) === f(b);
  });
```

forAll: Erlang QuickCheck

```
prop_json_encode_decode() ->
   ?FORALL(JSON_TERM,
        json_term(),
        begin
           JSON_TERM =:= decode(encode(JSON_TERM))
        end)
```

Breaking all the rules!!!

exists: existential quantifier
Can be proved

```
propContains = exists { c:Conference =>
(c.name ==“YOW!”) && (c.keywords.contains(“Awesome”)
  }
```

```
val propContains = exists { c:Conference =>
(c.name ==“YOW!”) && (c.keywords.contains(“Awesome”)
  }
```

+ OK, **proved** property.

Generally don't do this!

Patterns of Properties

Start with randomizing inputs for tests:
For any given input some result should be true.

"Fuzzing"

Move towards specification of model

Pattern: Relations

Pattern: Relations

```
propNewImplIsFaster = forAll{commands:
genListOfCommands =>
    time1 = time(FS.run(commands))
    time2 = time(CS.run(commands))
    time1 < time2
}
```

Pattern: Relations

```
propOptionsCost = forAll{
    option1: genOption, option2: genOption =>
        score1 = rank(option1)
        score2 = rank(option2)
        if (option1.cost <= option2.cost)
            score1 > score2
        else
            score1 < score2
}
```

Pattern: Reference Implementation

Pattern: Reference Implementation

```
forAll { list: List[String] =>
  bubbleSort(list) == quickSort(list)
}
```

Pattern: Reference Implementation

```
propNewImplSameResult = forAll{
    cmds: genListOfCommands =>
     state1 = FS.run_and_return_state(cmds)
     state2 = CS.run_and_return_state(cmds)
     assert.equal(state1,state2)
}
```

Pattern: Round Trip, Symmetry

Pattern: Round Trip

```
let revRevTree (xs:list<Tree>) =
    List.rev(List.rev xs) = xs
Check.Quick revRevTree
```

```
>OK, passed 100 tests
```

Pattern: Round Trip

```
propInsert = forAll{ person: genPerson =>
    p1 = svc.put(person)
    p2 = svc.get(person)
    assert.equal(p1,p2)
}
```

Pattern: Idempotent

```
propDBUpsertIsIdempotent =
    forAll{person : genPerson =>
     p1 = db.upsert(person)
     p2 = db.upsert(person)
     p3 = db.upsert(person)
     count = db.get(person).count()
     assert.equal(1,count)
}
```

Concurrency Tests?

Now we have to think harder about API design

Exceptions:

```
Prop.throws(classOf[IndexOutOfBoundsException])
```

Precondition - implication

Preconditions

```
notZeroProperty = forAll { num: Number =>
  (num != 0) ==> {
    ...do something that divides by num
  }
}
```

Preconditions

```
fakeRedHeadsProperty = forAll { p: Person =>
  (p.hair == “Red” && p.age > 49) ==> {
    ...test something for people with red hair aged 50+
  }
}
```

Writing all of the possible unit tests would be impossible

* Re-order already generated inputs & rerun function for commutative tests
* Nesting forall for relations

Specify

Generate

Shrink

All the basics are typically built in
but you often build your own

```
val myGen = for {
  n <- choose(1, 50)
 m <- choose(n, 2*n)
} yield (n, m)
```

```
listOf(choose(Some[genThing],None))
listOfN(8, arbitrary[Int])
```

Composable!

```
trait Color
case object Red extends Color
case object Black extends Color

trait Vehicle { def color: Color }
case class Mercedes(val color: Color) extends Vehicle
case class BMW(val color: Color) extends Vehicle
case class Tesla(val color: Color) extends Vehicle

val genColor = Gen.oneOf(Red, Black)

val genMerc = for {color <- genColor} yield Mercedes(color)
val genBMW = for {color <- genColor} yield BMW(color)
val genTesla = for {color <- genColor} yield Tesla(color)

val genNewCar = Gen.oneOf(genMerc, genBMW, genTesla)
```

```
let associativity (x:int) (f:int->float,g:float->char,h:char->int) =
    ((f >> g) >> h) x = (f >> (g >> h)) x
    Check.Quick associativity
```

Statistics

```
import org.scalacheck.Prop.{forAll, classify}
val p = forAll { n:Int =>
classify(n < 1000 , “small”, "large") {
  classify(n < 0, "negative", “positive”) {
    n+n == 2*n
   }
 }
}
```

* Standard edge cases
* Generating sequences of commands to test state

Generating commands?

* Lambdas - to execute each command
* Generate list of commands
* Check pre-conditions before executing each
* Execute and check state is as expected or that properties of the state machine hold true.

The generator can generate different things every time.

If you find a failure, consider making it a unit test for regression.

Specify
Generate
Shrink

Making diagnosis easy by reducing the counterexample

```
! Falsified after 2 passed tests.
 > ARG_0: -1
 > ARG_0_ORIGINAL: -1432
```

```
! Falsified after 20000 passed tests.
 > ARG_0: [“”]
 > ARG_0_ORIGINAL: [“A”,”YOW!”, “113”,”23234”,...]
```

You start with a failing example. Apply shrink function once to the generated failing input.

Then apply the property again. If it still fails repeat the process.

Otherwise you're shrunk.

If you create a generator,
maybe create a shrinker

* Multivariant tests with custom shrinkers?

Specify
Generate
Shrink

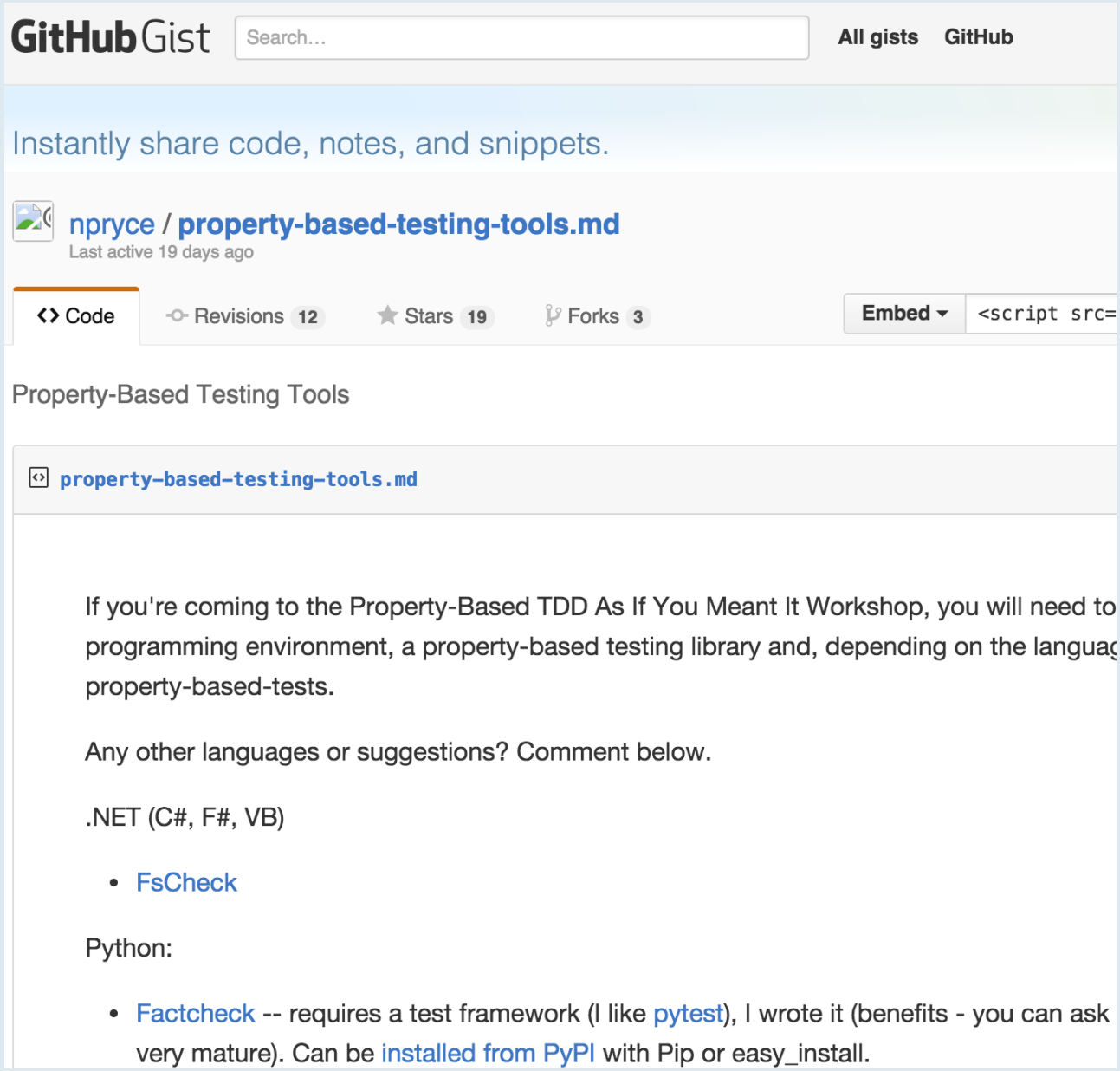
Configurations

* Minimum number of successful tests
* Maximum ratio between discarded and successful tests
* Minimum and maximum data size
* Random number generator
* Number of worker threads

Check all of the things

* HaskellQuickCheck
* Erlang : Quviq QuickCheck (commercial support) or Triq (Apache)
* C : Theft
* C++: QuickCheck++
* .NET (C#, F#, VB): FsCheck
* Ruby: Rantly
* Scala: ScalaCheck
* Clojure: ClojureCheck -- requires clojure.test
* Java: JavaQuickCheck -- requires JUnit or some other testing framework
* Groovy: Gruesome -- a quick and dirty implementation for Groovy
* JavaScript: QC.js, jsverify, claire
* Python:
    * Factcheck
    * Hypothesis
    * pytest-quickcheck - requires pytest, I found it hard to extend, and so wrote Factcheck

https://gist.github.com/npryce/4147916



"This all seems very obvious and logical"

Ranking Programs using Black Box Testing

Koen Claessen
Chalmers University of Technology
Gothenburg, Sweden.
koen@chalmers.se

John Hughes
Chalmers University of Technology and Quviq AB
Gothenburg, Sweden.
rjmh@chalmers.se

Michał Pałka
Chalmers University of Technology
Gothenburg, Sweden.
michal.palka@chalmers.se

Nick Smallbone
Chalmers University of Technology
Gothenburg, Sweden.
nicsma@chalmers.se

Hans Svensson
Chalmers University of Technology and Quviq AB
Gothenburg, Sweden.
hanssv@chalmers.se

Problem 1:

anonymize "pelle@foretag.se" ==
"p____@f_____.s_"

Problem 2:

type IntervalSet = [(Int,Int)], where for example the set
{1, 2, 3, 7, 8, 9, 10} would be represented by the list
[(1,3),(7,10)]

Problem 3:

SEND
MORE
MONEY

Map letters to digits, such that the calculation makes sense.
(In the example M=1,O=0,S=9,R=8,E=5,N=6,Y= 2, D = 7)

…We observed that all the QuickCheck test suites (when they were written) were **more effective** at detecting errors than any of the HUnit test suites.

…We observed that all the QuickCheck test suites (when they were written) were more effective at detecting errors than any of the HUnit test suites. … **Finally, we observed that QuickCheck users are less likely to write test code than HUnit users—even in a study of automated testing—suggesting perhaps that HUnit is easier to use**.

John Hughes of QuickCheck drew this!

A QuickCheck Example

Caesar's Cipher

$$e(x) = (x + k) \pmod{26}$$

```
char *caesar(int shift, char *input) {
  char *output = malloc(strlen(input));
  memset(output, '\0', strlen(input));

  for (int x = 0; x < strlen(input); x++) {
    if (isalpha(input[x])) {
      int c = toupper(input[x]);
      c = (((c - 65) + shift) % 26) + 65;
      output[x] = c;
    } else {
      output[x] = input[x];
    }
  }

  return output;
}
```

```haskell
caesar :: Int -> String -> String
caesar k = map f
  where
    f c
        | inRange ('A','Z') c = chr $ ord 'A' +
                                (ord c - ord 'A' + k) `mod` 26
        | otherwise = c
```

With a model we can check equivalence

```haskell
foreign import ccall "ceasar.h caesar"
    c_caesar :: CInt -> CString -> CString

native_caesar :: Int -> String -> IO String
native_caesar shift input = withCString input $ \c_str ->
    peekCString(c_caesar (fromIntegral shift) c_str)
```

```
$ ghci caesar.hs caesar.so
*Main> caesar 2 "ATTACKATDAWN"
"CVVCEMCVFCYP"
*Main> native_caesar 2 "ATTACKATDAWN"
"CVVCEMCVFCYP"
```

```
safeEquivalenceProperty = forAll genSafeString $ \str ->
      unsafeEq (native_caesar 2 str) (caesar 2 str)

deepCheck p = quickCheckWith stdArgs{ maxSuccess = 10000 } p
```

```
$ ghci caesar.hs caesar.so
*Main> deepCheck safeEquivalenceProperty
*** Failed! Falsifiable (after 57 tests):
"PMGDOSBUFYLIAITYVAPKZGTTWSCKMTXHJOKMYIEQFARLJGHJDPXSSXTP"
*Main> caesar 2
"PMGDOSBUFYLIAITYVAPKZGTTWSCKMTXHJOKMYIEQFARLJGHJDPXSSXTP"
"ROIFQUDWHANKCKVAXCRMBIVVYUEMOVZJLQMOAKGSHCTNLIJLFRZUUZVR"
*Main> native_caesar 2
"PMGDOSBUFYLIAITYVAPKZGTTWSCKMTXHJOKMYIEQFARLJGHJDPXSSXTP"
"ROIFQUDWHANKCKVAXCRMBIVVYUEMOVZJLQMOAKGSHCTNLIJLFRZUUZVR\SOH"
```

```
char *caesar(int shift, char *input) {
  char *output = malloc(strlen(input));
  memset(output, '\0', strlen(input));

  for (int x = 0; x < strlen(input); x++) {
    if (isalpha(input[x])) {
      int c = toupper(input[x]);
      c = (((c - 65) + shift) % 26) + 65;
      output[x] = c;
    } else {
      output[x] = input[x];
    }
  }

  return output;
}
```

```
char *caesar(int shift, char *input) {
  char *output = malloc(strlen(input));
  memset(output, '\0', strlen(input));

  for (int x = 0; x < strlen(input); x++) {
    if (isalpha(input[x])) {
      int c = toupper(input[x]);
      c = (((c - 65) + shift) % 26) + 65;
      output[x] = c;
    } else {
      output[x] = input[x];
    }
  }

  return output;
}
```

Improper Bound

```
char *caesar(int shift, char *input) {
  char *output = malloc(strlen(input));
  memset(output, '\0', strlen(input));

  for (int x = 0; x <= strlen(input); x++) {
    if (isalpha(input[x])) {
      int c = toupper(input[x]);
      c = (((c - 65) + shift) % 26) + 65;
      output[x] = c;
    } else {
      output[x] = input[x];
    }
  }

  return output;
}
```

```
$ ghci caesar.hs caesar.so
*Main> deepCheck safeEquivalenceProperty
+++ OK, passed 10000 tests.
```

John Hughes of QuickCheck drew this!

Where to go from here?

Google mature implementations for advice.

Final Thoughts

* TDD?
* Property tests live longer than unit tests
* Find different bugs
* Less code so more maintainable
    * More helper functions
    * More complex code
* Use in conjunction with other tests

Questions?

Amanda Laucher
@pandamonial
Pariveda Solutions Consultant
Mined Minds Co-Founder

Many thanks to @abedra