Beyond `xUnit` example-based testing: property-based testing with `ScalaCheck`

Franklin Chen

Pittsburgh Scala Meetup
April 11, 2013

Our goal: software correctness

Use all available tools that are *practical* in *context*.

Our goal: software correctness

Use all available tools that are *practical* in *context*.

- Static type system

Our goal: software correctness

Use all available tools that are *practical* in *context*.

- Static type system
  - We are Scala users!

Our goal: software correctness

Use all available tools that are *practical* in *context*.

- Static type system
  - We are Scala users!
- Two extremes

Our goal: software correctness

Use all available tools that are *practical* in *context*.

- Static type system
  - We are Scala users!
- Two extremes
  - Automated theorem proving

Our goal: software correctness

Use all available tools that are *practical* in *context*.

- Static type system
  - We are Scala users!
- Two extremes
  - Automated theorem proving
  - Testing

Our goal: software correctness

Use all available tools that are *practical* in *context*.

- Static type system
  - We are Scala users!
- Two extremes
  - Automated theorem proving
  - Testing

What is *practical*?

Theorem proving

Example: list append

```
// Given an implementation in ListOps of:
def append[A](xs: List[A], ys: List[A]): List[A]
```

How prove this property *without a shadow of a doubt*?

```
// for all lists xs and ys
ListOps.append(xs, ys).length == xs.length + y.length
```

Not (yet) *practical* in most contexts.

(Scala compiler cannot prove this property.)

More powerful static type system: dependent types

Try using a more powerful language?

- Example: Idris language, based on Haskell

Type checks!

```
(++) : Vect A n -> Vect A m -> Vect A (n + m)
(++) Nil       ys = ys
(++) (x :: xs) ys = x :: xs ++ ys
```

Still research, however.

- Also: Scala libraries such as `shapeless` pushing the edge of Scala's type system.
- Not (yet) *practical* for most of us.

Testing

Goals?

Testing

Goals?

- Give up goal of mathematical guarantee of correctness.

Testing

Goals?

- Give up goal of mathematical guarantee of correctness.
- Try to gain *confidence* that our code might be “probably” or “mostly” correct.

Testing

Goals?

- Give up goal of mathematical guarantee of correctness.
- Try to gain *confidence* that our code might be "probably" or "mostly" correct.
- Still a young field. Recent report on the state of testing: SEI blog

Testing

Goals?

- Give up goal of mathematical guarantee of correctness.
- Try to gain *confidence* that our code might be "probably" or "mostly" correct.
- Still a young field. Recent report on the state of testing: SEI blog

Remember to be humble when testing; avoid overconfidence!

Example-based testing

- `xUnit` frameworks such as `JUnit`
- Hand-craft specific example scenarios, make assertions

Popular Scala test frameworks that support `xUnit` style testing:

Example-based testing

- `xUnit` frameworks such as `JUnit`
- Hand-craft specific example scenarios, make assertions

Popular Scala test frameworks that support `xUnit` style testing:

- `ScalaTest`

Example-based testing

- `xUnit` frameworks such as `JUnit`
- Hand-craft specific example scenarios, make assertions

Popular Scala test frameworks that support `xUnit` style testing:

- `ScalaTest`
- `specs2`

Example-based testing

- xUnit frameworks such as JUnit
- Hand-craft specific example scenarios, make assertions

Popular Scala test frameworks that support xUnit style testing:

- ScalaTest
- specs2

For concreteness: I test with specs2, with SBT.

Individual examples of list append

```
// Individually hand-crafted examples
ListOps.append(List(3), List(7, 2)).length must_==
  List(3).length + List(7, 2).length
ListOps.append(List(3, 4), List(7, 2)).length must_==
  List(3, 4).length + List(7, 2).length
// ...
```

- Tedious to set up each example, one by one.
- Are we confident that we listed all the relevant cases, including corner cases?

Refactoring to a data table

```
// Hand-crafted table of data, but logic is factored out
"xs"         | "ys"         |
List(3)      ! List(7, 2) |
List(3, 4) ! List(7, 2) |> {
  (xs, ys) =>
  ListOps.append(xs, ys).length must_==
    xs.length + ys.length
}
```

Note: latest JUnit supports parameterized tests also.

Property-based testing: introducing ScalaCheck

- ScalaCheck library
  - Port of Haskell QuickCheck
  - Use standalone, or within ScalaTest or specs2
- Write down what we really intend, a universal property!

```
prop {
  (xs: List[Int], ys: List[Int]) =>
  ListOps.append(xs, ys).length must_==
    xs.length + ys.length
}
```

ScalaCheck automatically generates 100 random test cases (the default number) and runs them successfully!

Property-based testing: fake theorem proving?

- Theorem proving is hard.

Property-based testing: fake theorem proving?

- Theorem proving is hard.
- `ScalaCheck` allows us to pretend we are theorem proving, with caveats:

Property-based testing: fake theorem proving?

- Theorem proving is hard.
- `ScalaCheck` allows us to pretend we are theorem proving, with caveats:
  - How random are the generated data?

Property-based testing: fake theorem proving?

- Theorem proving is hard.
- `ScalaCheck` allows us to pretend we are theorem proving, with caveats:
  - How random are the generated data?
  - How useful for our desired corner cases?

Property-based testing: fake theorem proving?

- Theorem proving is hard.
- `ScalaCheck` allows us to pretend we are theorem proving, with caveats:
  - How random are the generated data?
  - How useful for our desired corner cases?
  - How can we customize the generation?

Property-based testing: fake theorem proving?

- Theorem proving is hard.
- `ScalaCheck` allows us to pretend we are theorem proving, with caveats:
  - How random are the generated data?
  - How useful for our desired corner cases?
  - How can we customize the generation?
  - How reproducible is the test run?

Property-based testing: fake theorem proving?

- Theorem proving is hard.
- `ScalaCheck` allows us to pretend we are theorem proving, with caveats:
  - How random are the generated data?
  - How useful for our desired corner cases?
  - How can we customize the generation?
  - How reproducible is the test run?
  - How confident can we be in the coverage?

Property-based testing: fake theorem proving?

- Theorem proving is hard.
- `ScalaCheck` allows us to pretend we are theorem proving, with caveats:
  - How random are the generated data?
  - How useful for our desired corner cases?
  - How can we customize the generation?
  - How reproducible is the test run?
  - How confident can we be in the coverage?

Property-based testing is still an area of research.

Property-based testing: fake theorem proving?

- Theorem proving is hard.
- `ScalaCheck` allows us to pretend we are theorem proving, with caveats:
  - How random are the generated data?
  - How useful for our desired corner cases?
  - How can we customize the generation?
  - How reproducible is the test run?
  - How confident can we be in the coverage?

Property-based testing is still an area of research.
But it is already useful as it is.

A failing test reports a minimal counter-example

What is wrong with this property about multiplication and division?

```
prop {
  (x: Int, y: Int) => (x * y) / y must_== x
}
```

Output:

```
> test
ArithmeticException: A counter-example is [0, 0]:
  java.lang.ArithmeticException: / by zero
```

Oops, our test was sloppy!

Introducing conditional properties

Exclude dividing by 0: use *conditional* operator `==>` with `forAll`:

```
prop {
  x: Int =>
  forAll {
    y: Int =>
    (y != 0) ==> { (x * y) / y must_== x  }
  }
}
```

- Generate random `x`.
- Generate random `y`, but discard test case if conditions fails.

All good now?

Testing as an iterative process

- Still get a counter-example.
- Int overflow.

```
> test
A counter-example is [2, 1073741824]
  (after 2 tries - shrinked
   ('452994647' -> '2','-2147483648' -> '1073741824'))
'-2' is not equal to '2'
```

Writing and refining a property guides our understanding of the problem.

The joy of experimental testing

`ScalaCheck` discovered an unexpected edge case for us!

- Write a general property before coding.

The joy of experimental testing

`ScalaCheck` discovered an unexpected edge case for us!

- Write a general property before coding.
- Write the code.

The joy of experimental testing

`ScalaCheck` discovered an unexpected edge case for us!

- Write a general property before coding.
- Write the code.
- `ScalaCheck` gives counter-example.

The joy of experimental testing

`ScalaCheck` discovered an unexpected edge case for us!

- Write a general property before coding.
- Write the code.
- `ScalaCheck` gives counter-example.
- Choice:

The joy of experimental testing

`ScalaCheck` discovered an unexpected edge case for us!

- Write a general property before coding.
- Write the code.
- `ScalaCheck` gives counter-example.
- Choice:
  - Fix code.
  - Fix property.

The joy of experimental testing

`ScalaCheck` discovered an unexpected edge case for us!

- Write a general property before coding.
- Write the code.
- `ScalaCheck` gives counter-example.
- Choice:
  - Fix code.
  - Fix property.
- Run `ScalaCheck` again until test passes.

Choice 1: fix code

Assume we want our high-level property, of multiplying and dividing of integers, to hold.
Choose a different representation of integer in our application: replace `Int` with `BigInt`:

```
prop {
  x: BigInt =>
  forAll {
    y: BigInt =>
    (y != 0) ==> { (x * y) / y must_== x  }
  }
}
```

Success!

Choice 2: fix property

Assume we only care about a limited range of `Int`.
Use a *generator* `Gen.choose`, for example:

```
forAll(Gen.choose(0, 10000), Gen.choose(1, 10000)) {
  (x: Int, y: Int) => { (x * y) / y must_== x }
}
```

Success!

Introduction to custom generators: why?

Attempted property:

```
prop {
  (x: Int, y: Int, z: Int) =>
  (x < y && y < z) ==> x < z
}
```

Output:

```
> test
Gave up after only 28 passed tests.
  142 tests were discarded.
```

Reason: too many `x`, `y`, `z` test cases that did not satisfy the condition.

Introduction to custom generators: how?

Try to generate data likely to be useful for the property.
Use `arbitrary` generator, `Gen.choose` generator, and `for`-comprehension:

```
val orderedTriples = for {
  x <- arbitrary[Int]
  y <- Gen.choose(x + 1, Int.MaxValue)
  z <- Gen.choose(y + 1, Int.MaxValue)
} yield (x, y, z)

forAll(orderedTriples) {
  case (x, y, z) =>
  (x < y && y < z) ==> x < z
}
```

Success!

More complex custom generator: sorted lists

Assume we want to specify the behavior of a function that inserts an integer into a sorted list to return a new sorted list:

```
def insert(x: Int, xs: List[Int]): List[Int]
```

We first try

```
prop {
  (x: Int, xs: List[Int]) =>
  isSorted(xs) ==> isSorted(insert(x, xs))
}
```

where we have already defined

```
def isSorted(xs: List[Int]): Boolean
```

Not constrained enough

Too many generated lists are not sorted.

```
> test-only *ListCheckSpec
Gave up after only 10 passed tests.
  91 tests were discarded.
```

So let's write a custom generator `someSortedLists`, to use with the property

```
forAll(someSortedLists) {
  xs: List[Int] => prop {
    x: Int =>
    isSorted(xs) ==> isSorted(insert(x, xs))
  }
}
```

Custom generator for sorted lists

- Choose a list size.
- Choose a starting integer `x`.
- Choose a list with first element at least `x`.

```
val someSortedLists = for {
  size <- Gen.choose(0, 1000)
  x <- arbitrary[Int]
  xs <- sortedListsFromAtLeast(size, x)
} yield xs
```

Now implement our `sortedListsFromAtLeast`.

Generating the sorted list

```
/**
  @return generator of a sorted list of length size
          with first element >= x
 */
def sortedListsFromAtLeast(size: Int, x: Int):
    Gen[List[Int]] = {
  if (size == 0) {
    Nil
  }
  else {
    for {
      y <- Gen.choose(x, x+100)
      ys <- sortedListsFromAtLeast(size-1, y)
    } yield y::ys
  }
}
```

What is going on? Classifiers

Gather statistics about the nature of the generated data. One way: classifiers.

```
forAll(someSortedLists) {
  xs: List[Int] => classify(xs.length < 300,
                            "length 0-299") {
    classify(xs.length >= 300 && xs.length < 800,
             "length 300-799") {
      classify(xs.length >= 800,
               "length 800+") {
        prop {
          x: Int =>
          isSorted(xs) ==> isSorted(insert(x, xs))
        }
      }
    }
  }
}
```

Classifier output

Output:

```
> test-only *ListCheckSpec
[info] > Collected test data:
[info] 42% length 300-799
[info] 31% length 0-299
[info] 27% length 800+
```

Much more!

Examples not covered today.

- Built-in support for generating sized containers.

Much more!

Examples not covered today.

- Built-in support for generating sized containers.
- Specify custom frequencies when choosing alternatives.

Much more!

Examples not covered today.

- Built-in support for generating sized containers.
- Specify custom frequencies when choosing alternatives.
- Generate objects such as trees.

Much more!

Examples not covered today.

- Built-in support for generating sized containers.
- Specify custom frequencies when choosing alternatives.
- Generate objects such as trees.
- Generate functions.

Much more!

Examples not covered today.

- Built-in support for generating sized containers.
- Specify custom frequencies when choosing alternatives.
- Generate objects such as trees.
- Generate functions.
- Supply own random number generator.

Much more!

Examples not covered today.

- Built-in support for generating sized containers.
- Specify custom frequencies when choosing alternatives.
- Generate objects such as trees.
- Generate functions.
- Supply own random number generator.
- Stateful testing.

Warning: false sense of security

- Automated testing: good
- Lots and lots of tests: good?
- Fallacy of big data overoptimism

Testing is not proof: *absence of evidence* of bugs does not equal *evidence of absence* of bugs.

Other property-based testing tools

`ScalaCheck` has known limitations:

- Not always easy to write good generator.
- Random generation does not provide complete coverage.
- Tests do not give the same results when run again.

Alternatives:

- SmallCheck for Haskell
  - Apparently ported to Scala
- Active area of research

The TDD community

- `JUnit` incorporating some property-based ideas in the new feature called Theories
- Many other programming languages gaining property-based testing!
- Nat Pryce of Growing Obect-Oriented Software Guided by Tests giving workshops on property-based TDD

Conclusion

- Automated testing is good.
- Property-based testing: put into your toolbox.
  - If you use Scala: use `ScalaCheck`.
  - If you use Java: use `ScalaCheck`!
- Be aware of limitations of testing.
- Have fun!