Functions and Determinism in Property-based Testing

Erik Osheim (**@d6**)

stripe

who am i?

- **typelevel** member λ
- maintain **spire**, **cats**, and other scala libraries[1]
- interested in expressiveness **and** performance ◒
- support machine learning at **stripe**

code at: http://github.com/non

[1] ScalaCheck co-maintainer as of Monday!

what will this talk cover?

1. property-based testing overview ♜
2. scalacheck case studies ☏
3. dive into generators ☟
4. deterministic function generation λ
5. some take-aways about laws and generators ☼
6. enthusiasm for testing! ☀

overview

origin story

The paper that launched a thousand implementations:

"QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs" (ICFP 2000)

by Koen Claessen and John Hughes

Introduces properties and generators, with a mention of shrinking as well.

in all languages [2]

- Haskell: QuickCheck, SmallCheck, LeanCheck, Hedgehog
- Scala: ScalaCheck, Scalaprops, Sonic
- Python: Hypothesis
- Clojure: test.check
- Java: junit-quickcheck
- C: Theft
- Javascript: qc.js
- Rust: QuickCheck for Rust
- Go: Gopter

[2] Most languages have more than one library, this is a semi-curated subset.

the basic idea

The basic unit is a property:

- Essentially a function that returns Boolean
- Properties must be true for all valid inputs: $\forall x.P(x)$
- Thus, one false result disproves the property
- In most cases we can't exhaustively test a property

Property-based tests search for counter-examples.

false positives

Most properties are possibly true or definitely false.

```
 ∀x P(x) = ¬∃x ¬P(x)    -- predicate is true
¬∀x P(x) =  ∃x ¬P(x)    -- predicate is false
```

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Dijkstra (1970) "Notes On Structured Programming"

in practice

To test a property we:

— choose how many passing cases we want
— generate that many test cases[3]
— evaluate our property for each case

The property is falsified (test fails) if any test case fails.

[3] Hopefully the tests cases we generate are mostly distinct.

in practice

We can base our confidence on the number of passing test cases.

— Run fewer interactively to keep things snappy
— Run many more in CI (e.g. Travis) for peace of mind

100 test cases give us a bit of confidence…

…but with 10k we have (up to) 100x as much confidence.

case studies

case 1: archery

Archery is an immutable 2D R-Tree implementation.

http://github.com/meetup/archery

R-Trees were first proposed in:

"R-Trees: A Dynamic Index Structure for Spatial Searching"
Antonin Guttman (SIGMOD 1984)

We'll look at how Archery tests its core algorithm.

```scala
property("rtree.nearestK works") {
  forAll { (es: List[Entry[Int]], p: Point, k0: Int) =>
    val k = (k0 % 1000).abs // ensure k is 0-999
    val rt = build(es)      // build an RTree[Int]

    // map each geometry in `es` to its distance from `p`.
    // then sort and find the closest `k` geometries.
    val expected = es.map(_.geom.distance(p))
      .sorted.take(k).toVector

    // find the closest `k` geometries using the RTree
    val got = rt.nearestK(p, k).map(_.geom.distance(p))

    got shouldBe expected // these results should agree
  }
}
```

case 1: archery

Bugs this has caught, or could catch:

— Broken R-Tree generation (build)
— Broken searching (nearestK)
— Duplicate entries at same point ignored
— Searching for nearest 0 points could crash

case 2: jawn

Jawn is a pluggable JSON parser.

Used by other JSON libraries, such as Circe.

https://github.com/non/jawn

We'll look at how Jawn tests its parser/renderer.

```scala
property("idempotent parsing/rendering") {
  forAll { value1: JValue =>
    val json1 = CanonicalRenderer.render(value1)
    val value2 = JParser.parseFromString(json1).get
    val json2 = CanonicalRenderer.render(value2)

    json2 shouldBe json1
    json2.## shouldBe json1.##

    value1 shouldBe value2
    value1.## shouldBe value2.##

    parser.Util.withTemp(json1) { t =>
      JParser.parseFromFile(t).get shouldBe value2
    }
  }
}
```

case 2: jawn

Bugs this has caught, or could catch:

- assorted parser bugs
- assorted rendering bugs
- parse → render changes data (e.g. 9 vs 9.0)
- broken equality on JValue
- broken hashCode on JValue
- parsing from files and strings differs

case 3: spire

Spire is a numeric library for Scala which is intended to be generic, fast, and precise.

http://github.com/non/spire

In this example we'll look at Spire's Interval type.

```scala
def testUnop(
    f: Interval[Rational] => Interval[Rational],
    g: Rational => Rational): Unit = {
  forAll { (orig: Interval[Rational]) =>
    val modified = f(a)
    sample(orig, 100).forall { x =>
      modified.contains(g(x)) shouldBe True
    }
  }
}

property("sampled unop pow(2)")(testUnop(_.pow(2), _.pow(2)))
property("sampled unop pow(3)")(testUnop(_.pow(3), _.pow(3)))
```

case 4: CountVectorizer

Converts a text feature into counts, including N-grams.

We are testing code that identifies all N-grams in a text.

```
val iter = "abcde".iterator.map(_.toString)
val it = new NGramIterator(it, 2, 3, None)
it.toList // List(ab, bc, abc, cd, bcd, de, cde)
```

(This test is taken from a project at Stripe.)

```
"work with any sequence" in {
  forAll { (data: NGramIteratorData) =>
    val NGramIteratorData(str, minN, maxN) = data

    val expected = (minN to maxN).flatMap { n =>
      iter(str).sliding(n).map(_.mkString(""))
    }

    val it = new NGramIterator(iter(str), minN, maxN, _ + _)
    val actual = it.toList.sortBy(_.length)
    assert(actual == expected)
  }
}
```

case study summary

Types of tests we saw:

- — Laws (e.g. idempotence, associativity, etc.)
- — Parallel evaluation (e.g. fast vs slow-but-correct)
- — Spot checking (e.g. sampling inside the interval)
- — Just exercising the code
- — Forcing us to think a bit about how we test

what are we missing?

Consider:

```
property("idempotent parsing/rendering") {
  forAll { value1: JValue =>
    ...
  }
}
```

The big question: how did we get a JValue to begin with?

generators

⚠️ WARNING ⚠️

What follows is a simplified view of ScalaCheck.

The simplification ellides:

— efficiency concerns
— crufty, legacy API details
— important features which we don't need
— the larger ScalaCheck framework

Later we'll compare ScalaCheck's Gen with this one.

anatomy of a generator

The core idea: generate random values.

```scala
// given a source of randomness,
// produce an A value.
case class Gen[A](run: Rng => A)

object Gen {
  def const[A](a: A): Gen[A] = Gen(_ => a)
}
```

anatomy of a generator

The core idea: generate random values **reproducibly**.

```
// given a source of randomness,
// produce an A value and an updated source.
case class Gen[A](run: Rng => (A, Rng))

object Gen {
  def const[A](a: A): Gen[A] = Gen(r => (a, r))
}
```

anatomy of a generator

The core idea: generate random values **reproducibly**.

```
// given a source of randomness,
// produce an A value and an updated source.
case class Gen[A](run: Rng => (A, Rng))

object Gen {
  def const[A](a: A): Gen[A] = Gen(r => (a, r))
}
```

(pay no attention to the **state monad** behind the curtain.)

why determinism?

- easier to reason about what is happening
- ensure test code is independent of test
- reproducible tests and test cases
- concurrent/parallel test evaluation
- can break rules if needed (e.g. in the REPL)

what do you think?

anatomy of a random-number generator

Here's an RNG sufficient for demo purposes:

```
// donald knuth's 64-bit MMIX rng
case class Rng(seed: Long) {
  def next: Rng = Rng(
    seed *
    6364136223846793005L +
    1442695040888963407L)
}
```

anatomy of a random-number generator

Rng represents a position in an immutable sequence:

```
object Rng {
  def random: Rng = {
    val seed: Long = scala.util.Random.nextLong
    Rng(seed)
  }
}

val rng0 = Rng.random   // Rng(3121071518240402 36)
val rng1 = rng0.next    // Rng(2643567112438381067)
val rng2 = rng1.next    // Rng(3536375599844977214)
val rng3 = rng2.next    // Rng(-8652326046818176971)
// and so on...
```

anatomy of a random-number generator

It's a relatively small step to a stream of bytes:

```
val rng0 = Rng.random
val stream = Stream.iterate(rng0, 6)(r => r.next)
// Stream(Rng(1344895957756080708), ?)

val bytes = stream.map(r => r.seed.toByte)
bytes.toList
// List(4, 3, -42, -19, -8, -25)
```

simple generators

Simple generators can extract values from seeds:

```
val long: Gen[Long] =
  Gen(r => (r.seed, r.next))

val bool: Gen[Boolean] =
  Gen(r => (r.seed >= 0, r.next))

val char: Gen[Char] =
  Gen(r => (r.seed.toChar, r.next))
```

Notice we return r.next to move along the RNG sequence!

simple generators

Even some "simple" generators are a bit fancy:

```
// doubles in the range [0, 1).
// e.g. (-1L >>> 11) * const = 0.9999999999999999
val double: Gen[Double] =
  Gen { r =>
    val shifted = r.seed >>> 11             // upper 53-bits
    val const = 1.1102230246251565e-16 // magic number
    val x = shifted * const                 // 0.0 <= x < 1.0
    (x, r.next)
  }
```

simple generators

Boilerplate alert!

We're always returning r.next in addition to our value:

```
val bool: Gen[Boolean] =
  Gen(r => (r.seed >= 0, r.next))

val char: Gen[Char] =
  Gen(r => (r.seed.toChar, r.next))
```

Seems like we could be a bit more expressive, right?

introducing map

We can use a map method to remove this kind of boilerplate!

```
case class Gen[A](run: Rng => (A, Rng)) { self =>
  def map[B](f: A => B): Gen[B] =
    Gen { rng0 =>
      val (a, rng1) = self.run(rng0)
      (f(a), rng1)
    }
}
```

simple generators (revisited)

```
val bool: Gen[Boolean] =
  long.map(_ >= 0) // using the original long generator

val double: Gen[Double] =
  long.map(x => (x >>> 11) * 1.1102230246251565e-16)

def upTo(limit: Int): Gen[Int] =
  double.map(x => (x * limit).toInt) // 0 <= _ < n

def oneIn(chance: Int): Gen[Boolean] =
  upTo(chance).map(_ == 0) // true 1-in-chance times
```

testing it out!

```
case class Gen[A](apply: Rng => (A, Rng)) {

  ...

  // impure! only do these from a REPL!

  def sample: A =
    run(Rng.random)._1

  def take(n: Int): List[A] =
    (1 to n).map(_ => sample).toList
}
```

testing it out!

```
val d6 = upTo(6).map(_ + 1)  // uniform values 1-6
val samples = d6.take(10000) // 10k random samples

val histogram = samples.groupBy(x => x).mapValues(_.size)
histogram.toList.sorted.foreach(println)
// (1,1656)
// (2,1646)
// (3,1629)
// (4,1699)
// (5,1706)
// (6,1664)
```

simple generators (summary)

What did we learn so far?

— it works! (at least in the REPL)
— map and long are enough for simple generators
— our RNG is mostly implicit (usually a good thing!)
— we should be explicit about distribution and range
— requires relatively small kernel of functionality

simple generators (summary)

What have we left out so far?

— generators with type parameters
— generators that need more than 64-bits of entropy
— lists and other collections
— correctly threading RNG state
— functions and other exotic types

fancy generators

Let's start with a generator we need: Gen[List[A]].

(We need this to write a better take method on Gen.)

Our gameplan:

— use generators recursively
— thread RNG state through appropriately
— profit!

fancy generators

If we're careful, we can generate lists:

```scala
// generate a list of n random values
def fixedList[A](gen: Gen[A], n: Int): Gen[List[A]] =
  if (n <= 0) Gen.const(Nil)
  else Gen { rng0 =>
    val (head, rng1) = gen.run(rng0)
    val (tail, rng2) = fixedList(gen, n - 1).run(rng1)
    (head :: tail, rng2)
  }
```

fancy generators

And we can use these to get even fancier:

```
// generate a randomly-sized list of random values
def list[A](gen: Gen[A], sized: Gen[Int]): Gen[List[A]] =
  Gen { rng0 =>
    val (n, rng1) = sized.run(rng0)
    fixedList(gen, n).run(rng1)
  }
```

As before, it seems like we should be able to simplify.

introducing flatMap

Did you see this coming?

```scala
case class Gen[A](run: Rng => (A, Rng)) { self =>

  ...

  def flatMap[B](f: A => Gen[B]): Gen[B] =
    Gen { rng0 =>
      val (a, rng1) = self.run(rng0)
      val gb: Gen[B] = f(a)
      gb.run(rng1)
    }
}
```

fancy generators (revisited)

```scala
// generate a list of n random values
def fixedList[A](gen: Gen[A], n: Int): Gen[List[A]] =
  if (n <= 0) Gen.const(Nil)
  else gen.flatMap { a =>
    fixedList(gen, n - 1).map(as => a :: as)
  }

  // i.e.
  // for {
  //   a  <- gen
  //   as <- fixedList(gen, n - 1)
  // } yield a :: as
```

fancy generators (revisited)

This one gets even nicer:

```
def list[A](gen: Gen[A], sized: Gen[Int]): Gen[List[A]] =
  sized.flatMap(n => fixedList(gen, n))
```

flatMap unlocks the power of A => Gen[B] methods:

```
def upTo(n: Int): Gen[Int]

def oneIn(n: Int): Gen[Boolean]
```

even more generators

```scala
def option[A](g: Gen[A]): Gen[Option[A]] =
  oneIn(10).flatMap { isNone =>        // 10/90% none/some
    if (isNone) Gen.const(None) else g.map(a => Some(a))
  }

def either[A, B](ga: Gen[A], gb: Gen[B]): Gen[Either[A, B]] =
  oneIn(2).flatMap {                   // 50/50% left/right
    case true => gb.map(b => Right(b))
    case false => ga.map(a => Left(a))
  }

def pair[A, B](ga: Gen[A], gb: Gen[B]): Gen[(A, B)] =
  ga.flatMap(a => gb.map(b => (a, b)))
```

so many generators!

```
def set[A](g: Gen[A]): Gen[Set[A]] =
  list(g, upTo(64)).map(_.toSet)

def vector[A](g: Gen[A]): Gen[Vector[A]] =
  list(g, upTo(64)).map(_.toVector)

val string: Gen[String] =
  list(char, upTo(32)).map(_.mkString)

def map[A](g: Gen[A]): Gen[Map[String, A]] =
  list(pair(string, g), upTo(64)).map(pairs => pairs.toMap)
```

fancy generator (summary)

What did we learn this time?

- flatMap is amazingly powerful! ✴
- we built product types (e.g. tuples, case classes)
- we built sum types (e.g. either, option)
- we built collections (e.g. set, vector, map)
- is there anything we can't do? ♫

a challenger appears!

What about `Gen[A => B]`?

Can we write a generator for function values?

a challenger appears!

What about `Gen[A => B]`?

Can we write a generator for function values?

What do you think?

putting the lazy in fp

Here's one that is technically "correct":

```
def constFunction[A, B](gb: Gen[B]): Gen[A => B] =
  gb.map { b =>
    (a: A) => b
  }
```

(But we only generate constant functions!)

putting the lazy in fp

```
val function: Int => Double =
  constFunction(double).sample

val values = (1 to 100).map(function)
values.toSet // Set(0.6081705385711283)
```

Unfortunately, these aren't very useful.

Let's try again.

principles are for other people

```
def wildFunction[A, B](gb: Gen[B]): Gen[A => B] = {

  // HACK: sample uses a random Rng value
  def wild(a: A): B = gb.sample

  Gen.const(wild)
}
```

At least they aren't constant functions!

principles are for other people

Let's see:

```
val function: Int => Boolean =
  wildFunction(bool).sample

val values = (1 to 5).map(_ => function(0))
// Vector(true, true, false, true, false)
```

They aren't constant functions,
because they aren't functions at all! ☠

hmmmm.

Are we stuck?

taking a step back

In both cases, we required a Gen[B].

But we don't have anything mentioning A.
(We don't need Gen[A]; we won't generate A values.)

What gives?

taking a step back

Recall, that Gen[B] is basically:

```
Rng => (B, Rng) // consume rng state to generate B
```

taking a step back

Recall, that Gen[B] is basically:

```
Rng => (B, Rng) // consume rng state to generate B
```

We sort of want the opposite, right?

```
??????????????? // ???????????????????????????????
```

taking a step back

Recall, that Gen[B] is basically:

```
Rng => (B, Rng) // consume rng state to generate B
```

We sort of want the opposite, right?

```
??????????????? // ???????????????????????????????
```

When in doubt, reverse things!

taking a step back

Recall, that Gen[B] is basically:

```
Rng => (B, Rng) // consume rng state to generate B
```

We sort of want the opposite, right?

```
??????????????? // consume A to generate rng state
```

When in doubt, reverse things!

taking a step back

Recall, that Gen[B] is basically:

```
Rng => (B, Rng) // consume rng state to generate B
```

We sort of want the opposite, right?

```
(A, Rng) => Rng // consume A to generate rng state
```

When in doubt, reverse things!

leap of faith

It's not totally clear yet, but let's go with it!

```scala
case class Cogen[A](rewind: (A, Rng) => Rng)

val clong: Cogen[Long] =
  Cogen { (n, rng0) =>
    val rng1 = Rng(rng0.seed ^ n) // xor n with the seed
    rng1.next        // get the next value in the sequence
  }
```

So now what?

explore the space

Can we plug these things together?

```
val cogen: Cogen[Long] = clong
val gen: Gen[Bool] = bool

def combined(rng0: Rng, n: Long): Bool = {
  val rng1 = cogen.rewind(n, rng0)
  gen.run(rng1)._1
}
```

Interesting... let's keep going!

explore the space

```
// (Rng, Long) => Bool
def combined(rng0: Rng, n: Long): Boolean = {
  val rng1 = cogen.rewind(n, rng0)
  gen.run(rng1)._1
}
```

explore the space

```scala
// (Rng, Long) => Bool
def combined(rng0: Rng, n: Long): Boolean = {
  val rng1 = cogen.rewind(n, rng0)
  gen.run(rng1)._1
}

// curry it into Rng => (Long => Boolean)
def recombined(rng0: Rng): Long => Boolean =
  { (n: Long) =>
    val rng1 = cogen.rewind(n, rng0)
    gen.run(rng1)._1
  }
```

let's try it!

```
val inputs = (1L to 5L)

// make sure f is deterministic
val f = recombined(Rng.random) // generate a function
inputs.map(f)  // Vector(false, false, true, false, true)
inputs.map(f)  // Vector(false, false, true, false, true)

// see if g is distinct and deterministic
val g = recombined(Rng.random) // generate another one
inputs.map(g)  // Vector(false, true, false, true, false)
inputs.map(g)  // Vector(false, true, false, true, false)
```

It appears to work!

polishing it up

So, our working Gen[A => B] looks like this:

```
def function[A, B](ca: Cogen[A], gb: Gen[B]): Gen[A => B] =
  Gen { rng0 =>
    def f(a: A): B = {
      val rng1 = ca.rewind(a, rng0)
      gb.run(rng1)._1
    }
    (f, rng0.next)
  }
```

tying up loose ends

We have Cogen[Long], but how do we make others?

tying up loose ends

We have Cogen[Long], but how do we make others?

```
case class Cogen[A](rewind: (A, Rng) => Rng) {
  def contramap[Z](f: Z => A): Cogen[Z] =
    Cogen((z, rng) => rewind(f(z), rng))
}
```

tying up loose ends

We have Cogen[Long], but how do we make others?

```
case class Cogen[A](rewind: (A, Rng) => Rng) {
  def contramap[Z](f: Z => A): Cogen[Z] =
    Cogen((z, rng) => rewind(f(z), rng))
}

val cbool: Cogen[Boolean] =
  clong.contramap(b => if (b) 1L else 0L)
```

tying up loose ends

```
val cint: Cogen[Int] =
  clong.contramap(x => x.toLong)

val cdouble: Cogen[Double] =
  clong.contramap(java.lang.Double.doubleToLongBits)

def clist[A](ca: Cogen[A]): Cogen[List[A]] =
  Cogen { (as, r0) =>
    as.foldLeft(r0)((r, a) => ca.rewind(a, r))
  }
```

turn it up to 11

```
val cogen: Cogen[List[Int]] = clist(cint)
val gen: Gen[List[Double]] = fixedList(double, 2)

val f: List[Int] => List[Double] =
  function(cogen, gen).sample

f(List(1,2,3))   // List(0.5494955859425557, 0.2120041015556522)
f(List(4, 5, 6)) // List(0.28665305811674324, 0.4006829927716514)
f(Nil)           // List(0.711467620936595, 0.24249997986473848)
```

Back to the Real World™

jawn, revisited

```
val jnull    = Gen.const(JNull)
val jboolean = Gen.oneOf(JTrue, JFalse)
val jlong    = arbitrary[Long].map(LongNum(_))
val jdouble  = arbitrary[Double].filter(isFinite).map(DoubleNum(_))
val jstring  = arbitrary[String].map(JString(_))

// Totally unscientific atom frequencies.
val jatom: Gen[JAtom] =
  Gen.frequency(
    (1, jnull),
    (8, jboolean),
    (8, jlong),
    (8, jdouble),
    (16, jstring))
```

jawn, revisited

```scala
def jarray(lvl: Int): Gen[JArray] =
  Gen.containerOf[Array, JValue](jvalue(lvl + 1)).map(JArray(_))

def jitem(lvl: Int): Gen[(String, JValue)] =
  for { s <- arbitrary[String]; j <- jvalue(lvl) } yield (s, j)

def jobject(lvl: Int): Gen[JObject] =
  Gen.containerOf[Vector, (String, JValue)](jitem(lvl + 1))
    .map(JObject.fromSeq)

def jvalue(lvl: Int = 0): Gen[JValue] =
  if (lvl >= MaxLevel) jatom
  else Gen.frequency((16, jatom), (1, jarray(lvl)), (2, jobject(lvl)))
```

how does Gen really work?

ScalaCheck's Gen is a bit more complicated:

```
type Gen[A] = (Params, Rng) => R[A]

type Params = ... // currently just a "size" parameter

type R[A] = (Option[A], Rng, ...)
```

how does Gen really work?

ScalaCheck's Gen is a bit more complicated:

```
type Gen[A] = (Params, Rng) => R[A]

type Params = ... // currently just a "size" parameter

type R[A] = (Option[A], Rng, ...)
```

Wait, Option[A]?? What!??

the other shoe

ScalaCheck allows generators to fail.

This is used to support things like filtering:

```
val positiveInt: Gen[Int] =
  arbitrary[Int].filter(_ > 0)
```

This is looks useful, right?

a terrible price

There is a downside:

"Gave up after only 32 passed tests. 162 tests were discarded."

a terrible price

There is a downside:

"Gave up after only 32 passed tests. 162 tests were discarded."

When a generator returns None, ScalaCheck discards that case and starts over.

After enough discarded cases, ScalaCheck gives up on the property. ☃

a terrible price

Additionally, partial generators totally break `Gen[A => B]`.

We relied on `gen.run(r)` always producing a `B` value.

So what does ScalaCheck do?

so...?

As of 1.13.x, ScalaCheck's Gen instances avoid filter.

At times we will "spin" to try to get a value:

```
def doPureApply(p: P, seed: Seed, retries: Int = 100): Gen.R[T] = {
  @tailrec def loop(r: Gen.R[T], i: Int): Gen.R[T] =
    if (r.retrieve.isDefined) r
    else if (i > 0) loop(doApply(p, r.seed), i - 1)
    else throw new Gen.RetrievalError()
  loop(doApply(p, seed), retries)
}
```

Worst-case: we have to fail (or throw). ☂

recommendations

Since filter leads to the most annoying ScalaCheck error (probably), and also breaks function generation[4]:

— Use existing combinators, e.g. Gen.choose(1, x)

— Avoid filter when possible.

— If necessary, considering mapping to valid values.

— Minimize the % of discarded values.

[4] It also makes collection generators much more likely to fail.


conclusions

what we saw

1. Generators aren't that complicated (in theory)
2. (Including function generators!)
3. Determinism is important
4. Pure functional programming can make things easier
5. You could roll your own property-based tests

what we heard

1. Writing properties is just writing tests, abstracted
2. Try to maximize the coverage/energy ratio
3. Pay attention to generator distribution and range
4. Avoid filter when possible
5. Don't be afraid to build custom generators ✂

what we did not cover

1. Shrinking (in any form)
2. Cases where we can be exhaustive
3. Managing recursive generation depth
4. "Approximate" laws (as seen in Algebird)
5. Type-level combinators (e.g. scalacheck-shapeless)
6. Detailed ScalaCheck walkthrough

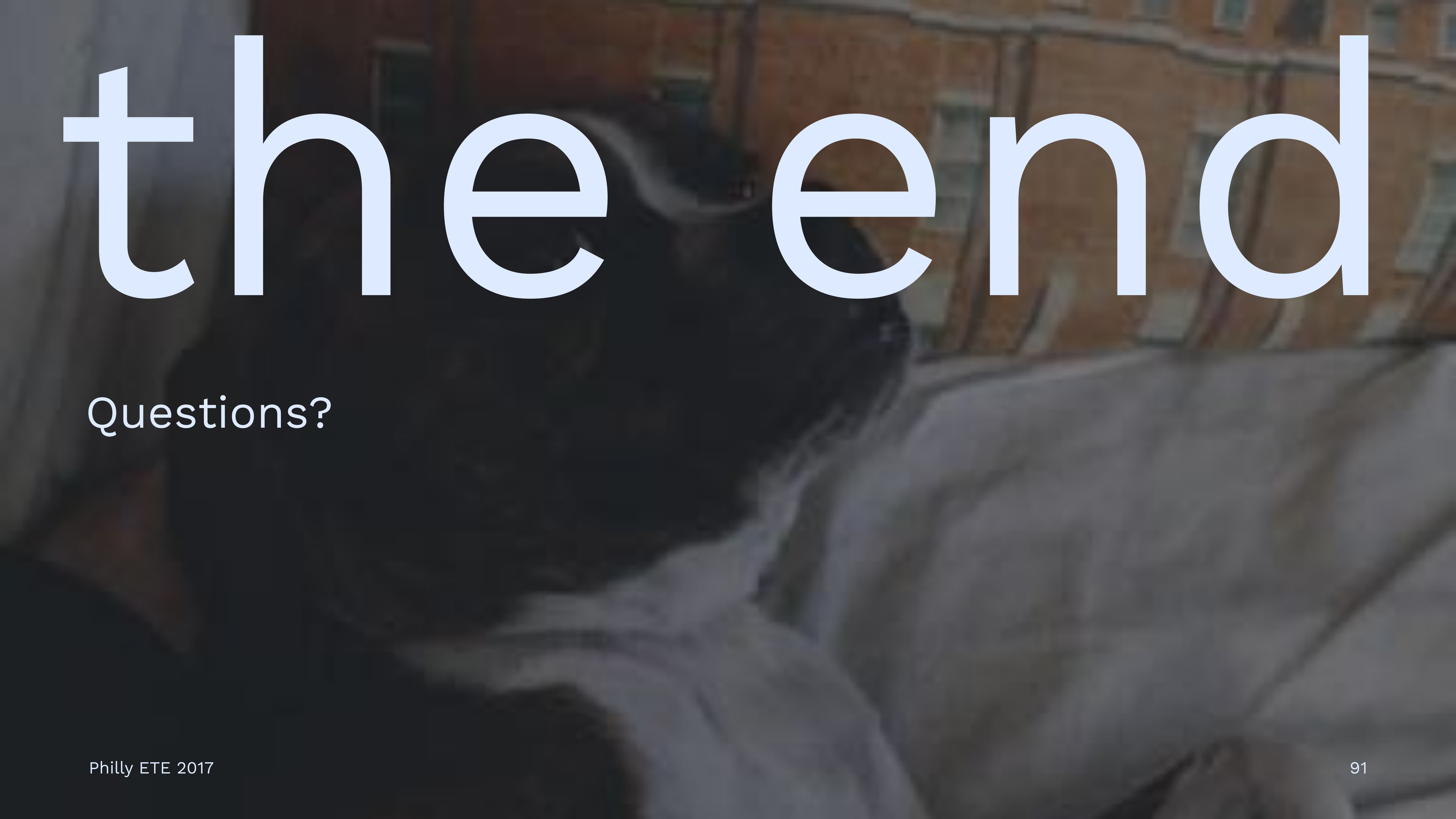
special thanks

ScalaCheck would not exist without Rickard Nilsson.

ScalaCheck would not have working function generators without the assistance of Kenji Yoshida.

ScalaCheck could not progress without the time and energy of its users and contributors.

☀ **Thank You!** ☀



the end

Questions?