Hands-on
ScalaCheck

Rickard Nilsson

The Third Halmstad Summer School on Testing
June 5 2013

Why ScalaCheck?

Haskell is great!

QuickCheck is great!

Scala seems interesting...

Let's port QuickCheck to Scala!

This presentation

11:00 Introduction / simple generators

11:15 Exercise session 1

11:30 Complex generators

11:50 Exercise session 2

12:20 Wrap-up

Generators

A generator is a function:

```
Gen.Params => Option[T]
```

… wrapped in the monadic type **`Gen[T]`**

Generator combinators

The module **`Gen`** contains building blocks for creating new generators

Some simple combinators:

```
Gen.value("Hello") : Gen[String]

Gen.choose(10,30)  : Gen[Int]

Gen.alphaChar      : Gen[Char]
```

Sampling generators

Each generator has a **`sample()`** method for convenience:

```
scala> Gen.value("Hello").sample
res0: Option[String] = Some(Hello)

scala> Gen.choose(10,30).sample
res1: Option[Int] = Some(13)

scala> Gen.alphaChar.sample
res2: Option[Char] = Some(q)
```

Combining generators

Scala's **`for`**-comprehension allows us to compose complex generators out of simple combinators:

```
val pair: Gen[(Int,Int)] = for {
  x1 <- Gen.choose(0,100)
  x2 <- Gen.choose(x1,2*x1)
} yield (x1,x2)

scala> pair.sample
res0: Option[(Int, Int)] = Some((43,78))
```

Higher-level generators

There are also generator combinators that takes other generators as arguments:

```
val pairs = Gen.listOf(pair)

scala> pairs.sample
res1: Option[List[(Int, Int)]] =
Some(List((54,94), (39,71), (67,116),
(48,51), (22,40), (100,120), (89,142),
(69,103), (52,104), (13,24), (71,134),
(75,133), (90,123), ...
```

Using generators in properties

Explicit use:

```
Prop.forAll(pair) {
  case (x,y) => y >= x
}
```

Implicit use:

```
Prop.forAll { (s: String, t: String) =>
  s.replace(s,t) == t
}
```

Implicit generators

You can use implicit generators as part of your own generators with the **`arbitrary[T]`** combinator

```
val palindrome: Gen[String] = for {
  s <- Arbitrary.arbitrary[String]
} yield s ++ s.reverse

val p = Prop.forAll(palindrome) { s =>
  s == s.reverse
}

scala> p.check
+ OK, passed 100 tests.
```

Implicit generators

For polymorphic generators, add the type constraint
`[T : Arbitrary]`

```
def container[T : Arbitrary]: Gen[List[T]] =
  for {
    t <- Arbitrary.arbitrary[T]
  } yield List(t)
```

Exercise session 1

Download zip-archive with exercises + slides here:

http://bit.ly/14tlK6z

One Scala module per exercise set, instructions in the code:

```
session1/src/main/E01.scala
session1/src/main/E02.scala
```

Look at the API docs for the Gen module:

http://rickynils.github.io/scalacheck/api-1.10.1/index.html#org.scalacheck.Gen$

Exercise session 1

The sbt build tool has a nifty console feature:

```
$ cd session1

$ sbt console

scala> E01.hello()
Hello Exercise 1!

scala> :type org.scalacheck.Gen.alphaChar
org.scalacheck.Gen[Char]
```

`case class` trick

ScalaCheck can generate the result of a function, by generating arbitrary parameters. Since each **`case class`** is a function, we can get generators for free:

```
val genPerson = Gen.resultOf(Person)

scala> genPerson.sample
res0: Option[Person] =
  Some(Person(„□□ ,□ø □□□□□□□□□□□□.. □□□₂□,-1))
```

Recursive generators

```
trait Tree[T] { def size: Int }

case class Leaf[T](
  item: T
) extends Tree[T] {
  def size = 1
}

case class Node[T] (
  children: List[Tree[T]]
) extends Tree[T] {
  def size = children.map(_.size).sum
}
```

Recursive generators

First try:

```
def genTree[T : Arbitrary]: Gen[Tree[T]] =
  Gen.oneOf(genLeaf[T], genNode[T])

def genLeaf[T : Arbitrary]: Gen[Leaf[T]] =
  for {
    item <-Arbitrary.arbitrary[T]
  } yield Leaf(item)

def genNode[T : Arbitrary]: Gen[Node[T]] =
  for {
    children <- Gen.listOf(genTree)
  } yield Node(children)
```

Recursive generators

Explosion!

```
scala> val intGenTree = genTree[Int]

java.lang.StackOverflowError
at org.scalacheck.Gen$class.$init$(Gen.scala:74)
at org.scalacheck.Gen$$anon$3.<init>(Gen.scala:232)
at org.scalacheck.Gen$.apply(Gen.scala:232)
at org.scalacheck.Gen$class.map(Gen.scala:96)
at org.scalacheck.Gen$$anon$3.map(Gen.scala:232)
at T$.genLeaf(<console>:20)
at T$.genTree(<console>:16)
...
```

Recursive generators

Lazy fix:

```
def genTree[T : Arbitrary]: Gen[Tree[T]] =
  Gen.lzy(Gen.oneOf(genLeaf[T], genNode[T]))

def genLeaf[T : Arbitrary]: Gen[Leaf[T]] =
  for {
    item <-Arbitrary.arbitrary[T]
  } yield Leaf(item)

def genNode[T : Arbitrary]: Gen[Node[T]] =
  for {
    children <- Gen.listOf(genTree)
  } yield Node(children)
```

Recursive generators

Still problems!

```
scala> val genIntTree = genTree[Int]

scala> genIntTree.sample

java.lang.StackOverflowError
   at org.scalacheck.Choose$$anon$5$
$anonfun$choose$3.apply(Gen.scala:27)
   at org.scalacheck.Gen$
$anonfun$parameterized$1.apply(Gen.scala:284)
   at org.scalacheck.Gen$
$anonfun$parameterized$1.apply(Gen.scala:284)
   at org.scalacheck.Gen$
...
```

Generator size

Each time ScalaCheck evaluates a generator, a **`size`** parameter is provided.

```
def listOf[T](g: Gen[T]) = Gen.sized { size =>
  for {
    n <- Gen.choose(0,size)
    l <- Gen.listOfN(n,g)
  } yield l
}
```

Recursive generators

A new try:

```
def genNode[T : Arbitrary]: Gen[Node[T]] =
  Gen.sized { size =>
    for {
      s <- Gen.choose(0,size)
      g = Gen.resize(size/(s+1),genTree)
      children <- Gen.listOfN(s,g)
    } yield Node(children)
  }
```

Recursive generators

A new try:

```
scala> genTree[Int].sample
res0: Option[Tree[Int]] =
Some(Node(List(Node(List(Node(List(Leaf(2147483647)
))))), Leaf(0), Node(List()), Leaf(-554792310),
Node(List(Node(List()), Leaf(0))),
Node(List(Node(List()), Node(List())))), Leaf(-
455164233), Leaf(2147483647), ...
```

Recursive generators

Another solution (not recommended in this case):

```
def genTree[T : Arbitrary]: Gen[Tree[T]] =
  Gen.lzy(
    Gen.frequency(
      (9, genLeaf[T]),
      (1, genNode[T])
    )
  )
```

Property preconditions

You can use preconditions in your properties

```
import org.scalacheck.Prop.BooleanOperators

Prop.forAll { (l1: List[Int], l2: List[Int]) =>

  l1.containsSlice(l2) ==> {
    val i = l1.indexOfSlice(l2)
    l1.slice(i, i+l2.length) == l2
  }

}
```

Property preconditions

… but ScalaCheck might not be able to fulfill your wish

```
scala> p.check
! Gave up after only 4 passed tests. 97 tests were
discarded.
```

Property preconditions

Solution: Create a generator!

```
def g[T : Arbitrary]: Gen[(List[T],List[T])] =
  for {
    l1 <- Gen.listOf(Arbitrary.arbitrary[T])
    n <- Gen.choose(0, l1.length)
    i <- Gen.choose(0, l1.length - n)
  } yield (l1, l1.slice(i,n))
```

Property preconditions

Solution: Create a generator!

```
Prop.forAll(g[Int]) { case (l1, l2) =>

  l1.containsSlice(l2) ==> {
    val i = l1.indexOfSlice(l2)
    l1.slice(i, i+l2.length) == l2
  }

}
```

Property preconditions

Solution: Create a generator!

```
scala> p.check
+ OK, passed 100 tests.
```

Remember, keep your precondition even if you use a custom generator!

Generator filters

You can put conditions on generators

```
val genOddNum = for {
  n <- Arbitrary.arbitrary[Int] if n % 2 == 1
} yield n

val genEvenNum =
  Gen.choose(-100,100).filter(_ % 2 == 0)
```

Generator filters

… but they can get you into the same problems as preconditions

```
scala> genOddNum.sample
res23: Option[Int] = Some(1)

scala> genEvenNum.sample
res24: Option[Int] = None

scala> genEvenNum.sample
res27: Option[Int] = Some(-12)
```

Implicit generators

You can add implicit generator support for your own types by defining an implicit instance of the **`Arbitrary[T]`** type

```
case class Person(
  firstName: String, lastName: String,
  birthYear: Int
)

val genPerson = for {
  firstName <- Gen.oneOf(”Mary”, ”John”)
  lastName  <- Gen.oneOf(”Smith”, ”Johnson”)
  birthYear <- Gen.choose(1900,2013)
} yield Person(firstName,lastName,birthYear)
```

Implicit generators

You can add implicit generator support for your own types by defining an implicit instance of the **`Arbitrary[T]`** type

```
implicit val arbPerson = Arbitrary[Person](genPerson)

val prop = Prop.forAll { p: Person =>
  p.birthYear >= 1900
}

scala> prop.check
+ OK, passed 100 tests.

scala> Arbitrary.arbitrary[Person].sample
res0: Option[Person] = Some(Person(John,Smith,1955))
```

Exercise session 2

One Scala module per exercise set:

```
session2/src/main/E03.scala
```

Instructions in the code

Time-saving feature:

```
sbt ~console
```

Wrap-up

Upcoming book – ”ScalaCheck: The Definite Guide”
http://www.artima.com/shop/scalacheck

User guide, bug reports, mailing list, etc
https://github.com/rickynils/scalacheck

ScalaCheck properties in ScalaCheck
https://github.com/rickynils/scalacheck/tree/master/src/test/scala/org/scalacheck

ScalaCheck properties in the Scala compiler
https://github.com/scala/scala/tree/master/test/files/scalacheck

Contact: rickynils@gmail.com