DEBUGGING AND TESTING WITH SCALACHECK

Martina Seidl

2017/11/14

Some Words on Scala

- Scala is **object-oriented**
  - every value is an object
  - classes and traits: types and behavior of objects
  - inheritance
- Scala is functional
  - every function is a value
  - anonymous functions
  - higher-order functions
  - support of currying
  - pattern matching
- Scala is statically typed
  - generic classes, polymorphic methods
  - variance annotations
  - upper and lower type bounds

Scala By Example I (from [4])

- class in Java:

```
1 public class Person {
2   public final String name;
3   public final int age;
4   Person(String name, int age) {
5       this.name = name;
6       this.age = age;
7   }
8 }
```

- class in Scala:

```
1 class Person(val name: String,
2              val age: Int) {}
```

Scala By Example I (from [4])

- filtering in Java (before Java 8):

```
1  Person[] people; Person[] minors; Person[] adults;
2  ...
3    ArrayList<Person> minorsList =
4                          new ArrayList<Person>();
5     ArrayList<Person> adultsList =
6                          new ArrayList<Person>();
7     for (int i = 0; i < people.length; i++)
8         (people[i].age < 18 ? minorsList :
9            adultsList).add(people[i]);
10    minors = minorsList.toArray(people);
11    adults = adultsList.toArray(people);
```

- filtering in Scala:

```
1  val people: Array[Person]
2  val (minors, adults) =
3        people partition (_.age < 18)
```

Scala By Example II (adapted from [1])

```
1  def sort(xs: Array[Int]) {
2
3    def swap(i: Int, j: Int) {
4      val t = xs(i); xs(i) = xs(j); xs(j) = t
5    }
6
7    def sort1(l: Int, r: Int) {
8      ...
9    }
10
11   sort1(0, xs.length - 1)
12 }
```

Scala By Example II (adapted from [1])

```
1   def sort1(l: Int, r: Int) {
2
3     val pivot = xs((l + r) / 2)
4     var i = l; var j = r
5
6     while (i <= j) {
7       while (xs(i) < pivot) i += 1
8       while (xs(j) > pivot) j -= 1
9         if (i <= j) {
10          swap(i, j)
11          i += 1
12          j -= 1
13        }
14    }
15
16    if (j < r) sort1(i, r)
17  }
```

Scala By Example II (adapted from [1])

```
1  def sort(xs: Array[Int]): Array[Int] = {
2    if (xs.length <= 1) xs
3    else {
4
5      val pivot = xs(xs.length / 2)
6
7      Array.concat(
8          sort(xs filter (pivot <)),
9          xs filter (pivot ==),
10         sort(xs filter (pivot >)))
11   }
12 }
```

The Power of Properties

The specification of properties ...

- ... helps to understand what the program shall do
- ... helps to understand what the program actually does
- ... helps to talk about the program
- ... can help to find an algorithm
- ... is valuable for debugging

What does ScalaCheck do?

- **User**:
  - specification of properties which should always hold
  - definition of random data for testing properties
  - no worries about missed test cases
- **ScalaCheck**:
  - automatic generation of test cases
  - checking if properties hold
  - shrinking (minimization of failing test cases)

ScalaCheck is ...

- ... an automated, property based testing tool for Scala/Java
- ... an extended port of Haskell QuickCheck
- ... available at www.scalacheck.org

first example:

```
1 object MyProperties extends
2         Properties("MyProperties") {
3
4   property("same length") =
5     forAll { (a: [Int]) =>
6       a.length == sort(a).length
7   }
8 }
```

ScalaChecks Highlights

- automatic testing of properties
- automatic generation of test data
- precise control of test data generation
- automatic simplification of failing test cases
- support for stateful testing of command sequences
- simplification of failing command sequences
- direct testing of property object from the command

```
scala> import org.scalacheck.Prop.forAll
import org.scalacheck.Prop.forAll

scala> val overflow = forAll { (n: Int) => n > n-1 }
overflow: org.scalacheck.Prop = Prop
scala> overflow.check
! Falsified after 6 passed tests.
> ARG_0: -2147483648
```

Basic Concepts

- properties

  `org.scalacheck.Prop`

- generators

  `org.scalacheck.Gen`

- test runner

  `org.scalacheck.Test`

Property

- testable unit in ScalaCheck
- class: `org.scalacheck.Prop`
- generation:
  - specification of new property
  - combination of other properties
  - use specialized methods

```
scala> object StringProps extends Properties("String") {
  |
  | property("startsWith") = forAll ( (a:String, b:String) =>
    (a+b).startsWith(a))
  | }
defined module StringProps

scala> StringProps.check
+ String.startsWith: OK, passed 100 tests.
```

Universally Quantified Property (Forall Property)

- create property: `org.scalacheck.Prop.forAll`
  - in: function which returns Boolean or a property
  - out: property
- check property: call of `check` method

```
1 import org.scalacheck.Prop.forAll
2
3 val propReverseList = forAll {
4     l: List[String] =>
5         l.reverse.reverse == l }
6
7 val propConcatString = forAll {
8     (s1: String, s2: String) =>
9         (s1 + s2).endsWith(s2) }
```

Data Generator

- generation of test data for
  - custom data types
  - subsets of standard data types
- representation: `org.scalacheck.Gen`

```
1  val myGen = for {
2    n <- Gen.choose(10,20)
3    m <- Gen.choose(2*n, 500)
4  } yield (n,m)
5
6  val vowel = Gen.oneOf('A', 'E', 'I', 'O', 'U')
7
8  val vowel1 = Gen.frequency( (3, 'A'), (4, 'E'),
9                              (2, 'I'), (3, 'O'), (1, 'U') )
```

A Generator for Trees

```
1  sealed abstract class Tree
2  case class Node(left: Tree, right: Tree, v: Int)
       extends Tree
3  case object Leaf extends Tree
4
5  val genLeaf = const(Leaf)
6
7  val genNode = for {
8    v <- arbitrary[Int]
9    left <- genTree
10   right <- genTree
11 } yield Node(left, right, v)
12
13 def genTree: Gen[Tree] = oneOf(genLeaf, genNode)
```

Statistics on Test Data

- collect infos on created test data
- inspection of distribution
- only trivial test cases?

```
1 def ordered(l: List[Int]) = l == l.sort(_ > _)
2
3 val myProp = forAll { l: List[Int] =>
4   classify(ordered(l), "ordered") {
5     classify(l.length > 5, "large", "small") {
6       l.reverse.reverse == l
7 ...
```

```
scala> myProp.check
+ OK, passed 100 tests.
> Collected test data:
78% large 16% small, ordered 6% small
```

Conditional Properties

- sometimes specifications are implications
- implication operator
- restricts number of test cases
- problem: condition is hard or impossible to fulfill
- property does not only pass or fail, but could be undecided if implication condition does not get fulfilled.

```
1 property("firstElement") =
2   Prop.forAll {
3     (xs: List[Int]) => (xs.size > 0) ==>
4           (xs.head == xs(0))
5   }
```

Combining Properties

combine existing properties to new ones

```
val p1 = forAll(...)

val p2 = forAll(...)

val p3 = p1 && p2

val p4 = p1 || p2

val p5 = p1 == p2

val p6 = all(p1, p2) // same as p1 && p2

val p7 = atLeastOne(p1, p2) // same as p1 || p2
```

Test Case Execution

- Module `Test`
  - execution of the tests
  - generation of the arguments
  - evaluation of the properties
  - increase of size of test parameters
  - reports success (passed) after certain number of tries
- Testing parameters in `Test.Parameters`
  - number of times a property should be tested
  - size bounds of test data
  - number of tries in case of failure
  - callback
- Statistics in `Test.Result`
- test properties with `Test.check`

Test Case Minimisation

- ScalaCheck tries to shrink failing test cases before they are reported
- Default by `Prop.forAll`
- No shrinking: `Prop.forAllNoShrink`

```
1 val p1 = forAllNoShrink(arbitrary[List[Int]])(
2                         l => l == l.removeDuplicates)
```

```
counter example:
List(8, 0, -1, -3, -8, 8, 2, -10, 9, 1, -8)
```

```
1 val p3 = forAll( (l: List[Int]) =>
2                         l == l.removeDuplicates )
```

```
counter example: List(-5, -5)
```

Customized Shrinking

- Definition of custom shrinking methods is possible
- Implicit method which returns `Shrink`$[T]$ instance
- Important: instances get smaller (otherwise loops possible)

```
1 implicit def shrinkTuple2[T1,T2]
2       (implicit s1: Shrink[T1], s2: Shrink[T2]
3 ): Shrink[(T1,T2)] = Shrink { case (t1,t2) =>
4   (for(x1 <- shrink(t1)) yield (x1, t2)) append
5   (for(x2 <- shrink(t2)) yield (t1, x2))
6 }
```

State-Full Testing

- What about testing combinations of functions?
- Solution: `org.scalacheck.Commands`
- Example: Test the behavior of a counter

```
1 class Counter {
2   private var n = 0
3   def inc = n += 1
4   def dec = n -= 1
5   def get = n
6   def reset = n = 0
7 }
```

State-full Testing

```
1
2  object CounterSpecification extends Commands {
3  type State = Int
4  type Sut = Counter
5
6  def newSut(state: State): Sut = new Counter
7
8
9   case object Inc extends UnitCommand {
10      def run(sut: Sut): Unit = sut.inc
11      def nextState(state: State): State = state + 1
12      def preCondition(state: State): Boolean = true
13      def postCondition(state: State, success: Boolean):
            Prop = success
14    }
15
16
17 }
```

References

[1] M. Odersky. Scala By Example, EPFL, 2014

[2] ScalaCheck Project Site: www.scalacheck.org

[3] Scala Project Site. ScalaCheck,
http://www.scalacheck.org/

[4] M. Odersky. Scala: How to make best use of functions
and objects, Tutorial slides, ACM SAC 2010

[5] R. Nilsson. ScalaCheck UserGuide,
https://github.com/rickynils/scalacheck/blob/master/doc/UserGuide.md