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:

```java
public class Person {
    public final String name;
    public final int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

class in Scala:

```scala
class Person(val name: String,
             val age: Int) {
}
```
filtering in Java (before Java 8):

```
Person[] people; Person[] minors; Person[] adults;

ArrayList<Person> minorsList =
  new ArrayList<Person>();
ArrayList<Person> adultsList =
  new ArrayList<Person>();
for (int i = 0; i < people.length; i++)
  (people[i].age < 18 ? minorsList :
   adultsList).add(people[i]);
minors = minorsList.toArray(people);
adults = adultsList.toArray(people);
```

filtering in Scala:

```
val people: Array[Person]
val (minors, adults) =
  people partition (_ .age < 18)
```
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }

  def sort1(l: Int, r: Int) {
    ...
  }

  sort1(0, xs.length - 1)
}
def sort1(l: Int, r: Int) {

  val pivot = xs((l + r) / 2)
  var i = l; var j = r

  while (i <= j) {
    while (xs(i) < pivot) i += 1
    while (xs(j) > pivot) j -= 1
    if (i <= j) {
      swap(i, j)
      i += 1
      j -= 1
    }
  }

  if (j < r) sort1(i, r)
}
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot <)),
      xs filter (pivot ==),
      sort(xs filter (pivot >)))
  }
}
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:**

```scala
object MyProperties extends Properties("MyProperties") {
  property("same length") =
    forAll { (a: [Int]) =>
      a.length == sort(a).length
    }
}
```
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
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
scala> object StringProps extends Properties("String") {
   |   property("startsWith") = forAll ( (a:String, b:String) =>
   |     (a+b).startsWith(a))
   | }
defined module StringProps
```

```scala
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

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

val propReverseList = forAll { l: List[String] => l.reverse.reverse == l }

val propConcatString = forAll { (s1: String, s2: String) => (s1 + s2).endsWith(s2) }
```
Data Generator

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

```scala
val myGen = for {
  n ← Gen.choose(10,20)
  m ← Gen.choose(2*n, 500)
} yield (n,m)

val vowel = Gen.oneOf('A', 'E', 'I', 'O', 'U')

val vowel1 = Gen.frequency((3, 'A'), (4, 'E'),
                           (2, 'I'), (3, 'O'), (1, 'U'))
```
A Generator for Trees

sealed abstract class Tree

case class Node(left: Tree, right: Tree, v: Int) extends Tree

case object Leaf extends Tree

val genLeaf = const(Leaf)

val genNode = for {
  v <- arbitrary[Int]
  left <- genTree
  right <- genTree
} yield Node(left, right, v)

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         }
8       }
9
10  ...
```

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.

```scala
property("firstElement") = Prop.forAll {
  (xs: List[Int]) => (xs.size > 0) ==> (xs.head == xs(0))
}
```
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

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

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

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

counter example: List(-5, -5)
Customized Shrinking

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

```scala
implicit def shrinkTuple2[T1, T2]
  (implicit s1: Shrink[T1], s2: Shrink[T2])
  : Shrink[(T1, T2)] = Shrink {
    case (t1, t2) =>
      (for (x1 ← shrink(t1)) yield (x1, t2)) append
      (for (x2 ← shrink(t2)) yield (t1, x2))
  }
```
State-Full Testing

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

```scala
class Counter {
  private var n = 0
  def inc = n += 1
  def dec = n -= 1
  def get = n
  def reset = n = 0
}
```
object CounterSpecification extends Commands {
  type State = Int
  type Sut = Counter

  def newSut(state: State): Sut = new Counter

  case object Inc extends UnitCommand {
    def run(sut: Sut): Unit = sut.inc
    def nextState(state: State): State = state + 1
    def preCondition(state: State): Boolean = true
    def postCondition(state: State, success: Boolean): Prop = success
  }
}
References


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

